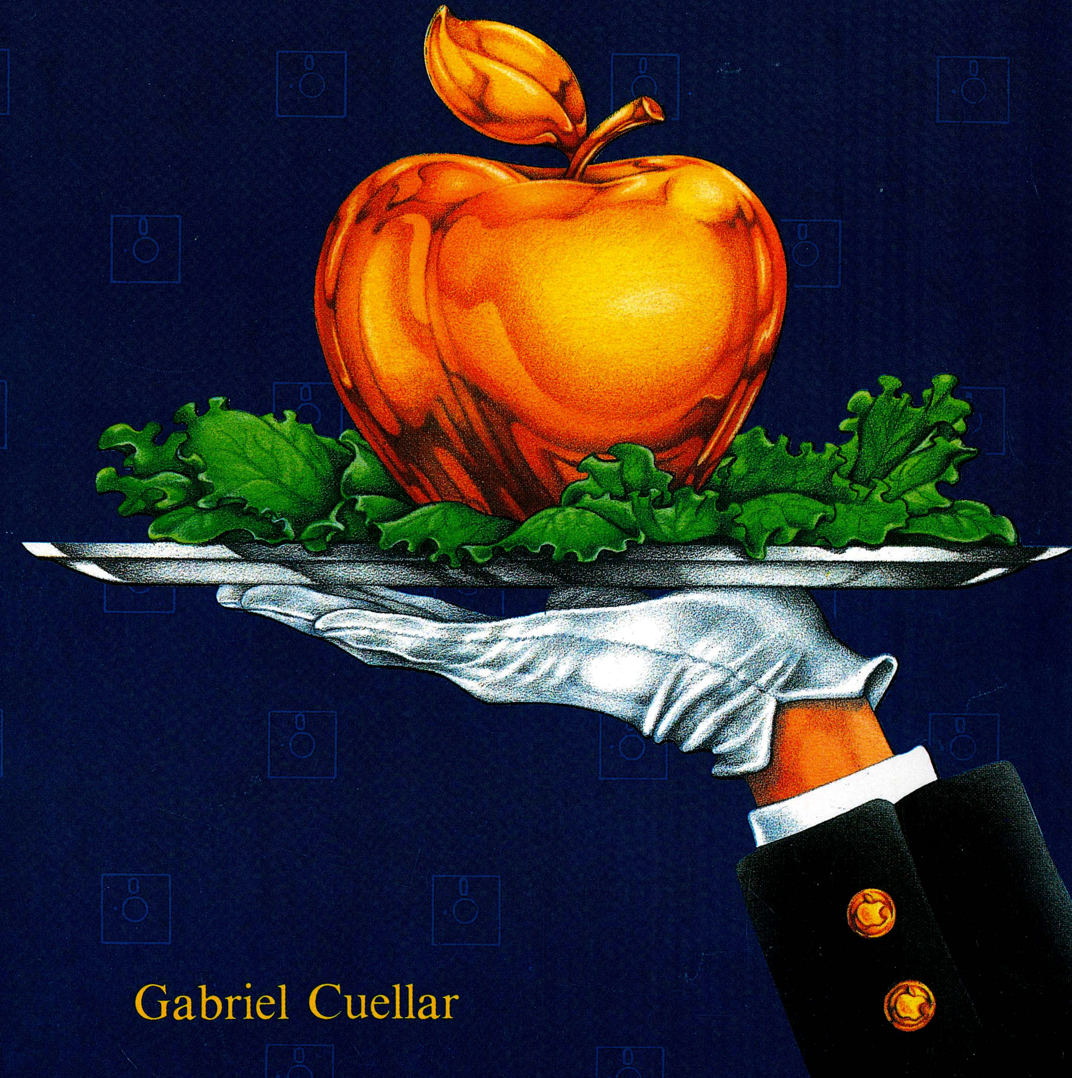
Reston

FANCY PROGRAMMING
IN APPLESOFT®



Gabriel Cuellar

Fancy Programming in Applesoft

Fancy Programming in Applesoft

Gabriel Cuellar

Reston Publishing Company
A Prentice-Hall Company
Reston, Virginia

Library of Congress Cataloging in Publication Data

Cuellar, Gabriel.
Fancy programming in Applesoft.

1. Apple II (Computer)—Programming. I. Title.
QA76.8.A662C8 1983 001.64'24 82-24110
ISBN 0-8359-1857-2 (disk)
ISBN 0-8359-1856-4 (pbk.)

© 1983 by Gabriel Cuellar

All rights reserved. No part of this book may be reproduced in any way, or by any means, without permission in writing from the publisher and the author.

10 9 8 7 6 5 4 3 2 1

Printed in the United States of America

To Tito, Estela, Enrique, and Mila

Contents

Acknowledgments xiii

Programs xv

Preface xix

1. Input 1

1.1 Rejecting the Unwanted Characters 2
1.2 Accepting the Backspace 2
1.3 Accepting Return 4
1.4 Accepting Only Numbers 4
1.5 Controlling the Length 5
1.6 Adding a Prompt Line 6
1.7 Continuing When the Length is Reached 7
1.8 Default Values 8
1.9 Deleting the Entire Line 9
1.10 Recuperating the Deleted Line 9
1.11 Returning to the Menu 12
1.12 Selective Input 13
1.13 Input without Stopping the Program 15
1.14 Input While Program Is Busy 17
1.15 Clearing the Keyboard 19

2. Output 21

2.1 Formatted Output 21
2.2 Formatting Strings 22
2.3 Formatting Numbers 23
2.4 Fixing the Length 23
2.5 Rounding 25
2.6 Format with Special Symbols 26
2.7 Formatting Very Large and Small Numbers 28
2.8 Output with Fixed Length 29

3. Sorting 31

3.1 Bubble Sort 32
3.2 Delayed Exchange Bubble Sort 33
3.3 Insertion Sort 34
3.4 Shell Sort 35
3.5 Tagfiles 37
3.6 Multiple Tagfiles 39
3.7 Binary Search 40
3.8 Pointers 42

4. Files 51

4.1 Sequential Files 51
4.2 Reading Separators 52
4.3 Reading Files of Unknown Length 53
4.4 Writing on Existing Files 53
4.5 Random Access Files 54
4.6 Keeping Files Open 55
4.7 Fixing the Length of the Records 55
4.8 Packing Data 58
4.9 Changing the Drive, Slot, and Volume 60
4.10 Maxfiles 61
4.11 Reading the Keyboard while Working on the Disk 61
4.12 Turning Off DOS 62

5. Exec Files 63

5.1 Plotting a Function 63
5.2 Data Creator 66
5.3 Program Modification 68
5.4 Additional Uses of Exec Files 74

6. Screen 75

6.1 Basic Switches 75
6.2 Video Screen 76
6.3 Copying the Screen 76
6.4 Displaying Low Resolution 77
6.5 High Resolution Screen 77
6.6 Activity Switch 78
6.7 Clearing High Resolution 78
6.8 Finding the HCOLOR 79
6.9 Switching Between Modes 79

7. Text 81

7.1 Printing the Text Screen 81
7.2 Changing the Screen Attribute 82
7.3 Scrolling the Screen 85
7.4 Reading Information from the Catalog 87
7.5 Very Low Resolution 88
7.6 Ten Characters 90
7.7 Using the Second Page of Text 93
7.8 Help Files 93
7.9 More than Two Pages in Text 95
7.10 Text Editor 96

8. Low Resolution 115

8.1 Saving Low Resolution Screens 115
8.2 Moving a Low Resolution Image 116

8.3 Low Resolution Editor 117
8.4 Using the Second Page of Low Resolution 124
8.5 More than Two Pages of Low Resolution 124

9. High Resolution 125

9.1 Protecting the High Resolution Pages 125
9.2 High Resolution Editor 125
9.3 Transferring Page 1 to Page 2 129
9.4 Pages 3, 4, and 5 of High Resolution 131
9.5 Section Scanner 132
9.6 Dot Organization 136
9.7 Scanner 137
9.8 Negative HI-RES 143
9.9 Combining Pages 1 and 2 146
9.10 Animation 149
9.11 Shapes 150
9.12 Creation of a Shape Table 151
9.13 Using Shapes 155
9.14 Shape Generator 156
9.15 Text in High Resolution 163
9.16 Seventy Characters 169
9.17 Inverse 171
9.18 Emulating the Print Command in High Resolution 172
9.19 Seventy by Thirty Two Characters 175
9.20 Generating the Alphabetic Shape Tables 175

10. Sound 181

10.1 Producing Sounds 181
10.2 Amplifying the Sounds 183
10.3 Producing Musical Notes 183
10.4 The Musical Scale 184
10.5 Computer Piano 185
10.6 Storing the Melodies 188

10.7 Music Entry 191
10.8 Harmony 196
10.9 Songs in Harmony 198

11. Utilities 201

11.1 Boolean Variables 201
11.2 Recovering from "New" 204
11.3 Creating Hidden Commands 206
11.4 A Very Powerful Hidden Tool 208
11.5 Hiding Characters in the Catalog 209
11.6 Hidden Character Creator 213
11.7 Relocating a Program 215
11.8 Having More than One Program in Memory 216
11.9 Saving an Applesoft Program as a Binary File 217
11.10 Controlling the Reset Key 218
11.11 Voluntary Garbage Collection 219
11.12 Binary Numbers 220
11.13 Calculating Equivalences 223
11.14 Subroutines that Call Themselves 227
11.15 Extended Precision Operations 230
11.16 Lower Case for the Printer 238

Index 243

Acknowledgments

I would like to thank Maria Emilia Correa for having introduced me to the world of computers, and Susie Jessup for her constant help and support.

Programs

	Page
REJECT	2
BACKSPACE	3
RETURN	4
ONLY NUMBERS	5
LENGTH CONTROL	5
PROMPT LINE	6
LENGTH CONTINUE	8
DEFAULT	9
DELETE LINE	10
RECUPERATE LINE	11
ESCAPE	12
SELECTIVE INPUT	14
NON-STOP	16
INPUT WHILE BUSY	18
NO FORMAT	22
ELEMENTARY FORMAT	24
ROUND	26
FORMAT WITH SYMBOLS	27
LARGE NUMBERS	29
FIXED LENGTH	30
SMALLEST	32
BUBBLE SORT	33
DELAYED EXCHANGE	33
INSERTION SORT	34
SHELL SHORT	36
TAGFILE	39

	Page
MULTIPLE TAGFILE	40
BINARY SEARCH	42
POINTERS	48
READ FILE	52
FIXED LENGTH RECORD	57
PACK DATA	60
PRIME NUMBERS	62
PLOTTER	64
FINAL PLOTTER	65
DATA CREATOR	67
MODIFIER	72
SCREEN DUMP	82
ATTRIBUTE	84
SCROLL	86
READ CATALOG	88
VERY LOW RESOLUTION	89
TEN CHARACTERS	91
HELP	95
EDITOR	107
LOW RESOLUTION EDITOR	122
HIGH RESOLUTION EDITOR	128
TRANSFER	130
SCANNER	134
BIT SCANNER	140
NEGATIVE	144
COMBINE	147
ANIMATION	150
SHAPE GENERATOR	161
FORTY CHARACTERS	167
SEVENTY CHARACTERS	170
INVERSE FORTY	172
INVERSE SEVENTY	172
EMULATOR FORTY	173
EMULATOR SEVENTY	174
ALPHABET CREATOR	176
ALPHABET 70 CREATOR	178
BURP DOWN	182
BURP UP	182
NOISE	182

	Page
RATTLE	182
SIREN	183
MUSIC	184
CASCADE	184
COMPUTER SONG	184
NOTES	185
PIANO	187
RECORDER	189
MUSIC ENTRY	194
BELLS	197
CHORDS	198
HIDDEN CATALOG	212
HIDDEN CHARACTER CREATOR	214
RELOCATE	216
RESET CONTROL	219
CONVERSION	225
RECURSIVE	230
EXTENDED PRECISION	236
LOWER CASE	240

Preface

This book was written for those who have already worked with Applesoft and feel they want to write more 'professional' programs, but realize that the information included in the Applesoft and DOS manuals is not enough, or not clear enough.

A tremendous amount of hardware and software available in the market make the Apple II (and Apple II plus) one of the most powerful and versatile personal computers in its price range. Nevertheless, this machine can be programmed to do virtually everything without leaving Applesoft or buying any additional hardware; only when speed or space become critical is there no other choice but to use a faster and more powerful language.

After reading this book you will be able to control most of the features of the Apple without ever having to go into the monitor (where you would have to program in machine language) or using any assembler at all. The areas covered—Input, Output, Text, Exec and Binary Files, Text Screen, Low and High Resolution Graphics, Sound, Data Sorting and Handling in general, and a variety of other topics—include most of what can make a program very difficult (and at times even impossible) to develop in Applesoft. There are a few areas in which Applesoft is incapable of achieving enough speed by itself (as in most of the programs that generate musical sounds) and a very short machine language subroutine is essential. Even in those cases, everything is handled by Applesoft; the routines are placed in memory using the POKE command, and their actions performed via a CALL to the address where the routines have been placed.

The book has been divided into 11 chapters, each of which does not depend on the previous ones. This feature lets you start reading anywhere you want. However, since some of the techniques explained can be essential in examples of different topics, you may have to study parts of other chapters to clearly understand them.

Since the book is oriented to those who write programs, for the sake of brevity many of the methods explained are not used in the sample programs (for example, very few of them have ONERR GOTO or use controlled INPUT). Likewise, since every line number of every program has been explained with great detail, very few REM statements have been included.

Fancy Programming in Applesoft

1

INPUT

When the INPUT command is used to fetch a number or a string from the keyboard, the program stops, and does not continue until a <return> is pressed. During that lapse, the program loses the control of the machine; if the response to the input is an expected type, the return to the program will be normal.

Unfortunately, there can be many irregularities in the responses to the command. Here is a list of some of them:

1. If the backspace is typed when the cursor is at the first position of the input line, the cursor jumps to the left margin of the next line; the INPUT is still active, but the result is very disorienting. In addition, the characters typed will now end up in the next line.
2. If control characters are typed, they are included in the input string, but are not displayed. If the variable used is of string type, neither the length nor the comparison with other strings will correspond with the expected result, because of these "invisible" characters. If the variable is of a numerical type, these characters will cause a 'REENTER' message to be PRINTed, and the INPUT to be repeated.
3. If more than 255 characters are typed, Applesoft PRINTs a backslash (\), ignores the line already typed, and starts the INPUT from the beginning.
4. If ctrl-C is the first character typed, the program will stop after <return> is pressed.
5. If a number is expected, a string will cause a 'REENTER' to be PRINTed and the INPUT to be repeated.

6. If a number is expected and more digits are typed than a variable can take, an 'OVERFLOW ERROR' will cause the program to terminate.
7. If <esc> is typed, the cursor commands can be used (See Appendix B, page 110, and Appendix J, page 130, of the Applesoft manual).
8. If a comma or a colon is typed, the characters following it will be ignored and the message 'EXTRA IGNORED' will be PRINTed.

This list is by no means exhaustive, but it can show you how much control you lose when an INPUT is executed. The goal of this chapter is to give you techniques to control those situations.

1.1 REJECTING THE UNWANTED CHARACTERS

Program REJECT reads one character at a time from the keyboard, using the GET statement in line 500. The character is analyzed and rejected if it is not displayable. To make this rejection noticeable, a bell character is PRINTed. Otherwise, the character is echoed and added to X$, which keeps a copy of the string of accepted characters.

Since every nondisplayable character is being rejected, the backspace cannot be used to correct mistakes, and there is no way to signal the end of the input with <return>.

```
1   REM   ** REJECT **
10 BL$ =  CHR$ (7): HOME : VTAB 10
500  GET W$: IF W$ < " " OR W$ > "^" THEN  PRINT
     BL$;: GOTO 500
510  PRINT W$;:X$ = X$ + W$: GOTO 500
```

1.2 ACCEPTING THE BACKSPACE

Program BACKSPACE GETs the characters in line 500. If the character is a displayable one, the program jumps to line 700, where it is echoed and added to X$.

If the character pressed is a backspace (CHR$ (8) or ctrl-H), lines 520 through 540 analyze three different cases:

1. There are no characters in X$.
   In this case the bell character is PRINTed, indicating an error.

2. There is only one character in X$.
   This case must be treated separately, since trying to evaluate LEFT$ (X$, 0) would cause an error (the numerical argument of the function LEFT$ must be >0 and <256). X$ is emptied, and B$ is PRINTed. See the explanation of BS$, below.

3. X$ has more than one character.
   This is considered the normal case, in which the last character of X$ is deleted (LEFT$ (X$, LEN (X$) – 1)), and the cursor position is updated using BS$.

If the character is neither backspace nor a displayable one, line 510 PRINTs the bell and ignores the character.

There are three characters in BS$: a backspace, a blank, and another backspace. When a backspace is PRINTed, it moves the cursor to the previous location on the screen, provided that it is not at the left-top corner, in which case nothing is done.

Therefore, the first backspace in BS$ moves the cursor to the previous location; a blank is PRINTed, erasing the character that was previously there, and a new backspace places the cursor in that clear location. The final result is to move the cursor backward and leave it standing on a blank.

```
1   REM   ** BACKSPACE **
10   HOME :BS$ =  CHR$ (8) + " " +  CHR$ (8):BL$ =
       CHR$ (7)
20   HTAB 4: VTAB 8
500   GET W$: IF W$ >  = " " AND W$ <  = "Z" THEN
      700
510   IF W$ <  >  CHR$ (8) THEN  PRINT BL$;: GOTO
      500
520   IF X$ = "" THEN  PRINT BL$;: GOTO 500
530   IF  LEN (X$) = 1 THEN X$ = "": PRINT BS$;: GOTO
      500
540 X$ =  LEFT$ (X$, LEN (X$) - 1): PRINT BS$;: GOTO
      500
700   PRINT W$;:X$ = X$ + W$: GOTO 500
```

1.3 ACCEPTING <RETURN>

The usual way to terminate an INPUT is by typing <return>. In program RETURN, line 20 asks for a name, and calls subroutine 450. Line 450 clears X$, which is going to be used to store the characters that are accepted by the input part of the subroutine.

Line 500 GETs the character and branches to line 700 if it is a displayable one. Otherwise, lines 520 through 540 take care of the backspace case.

If the character pressed is a <return>, line 550 RETURNs the subroutine. Notice that the actual CHR$ (13) is not included in X$.

If W$ was not trapped in any of the previous cases, it is an illegal character. Line 560 PRINTs a bell and ignores it.

Line 30 stores X$ in N$, and line 40 asks for an address: Subroutine 450 is called once more, and this time X$ is stored in A$. Line 40 asks for an address, and line 50 shows the two strings.

```
1   REM   ** RETURN **
10   HOME :BS$ =  CHR$ (8) + " " +  CHR$ (8):BL$ =
       CHR$ (7)
20   HTAB 4: VTAB 8: PRINT "NAME: ";
30   GOSUB 450:Q$ = X$
40   HTAB 4: VTAB 12: PRINT "ADDRESS:";: GOSUB 450
      :A$ = X$
50   HOME : PRINT Q$: PRINT A$: END
450 X$ = ""
500  GET W$: IF W$ >  = " " AND W$ <  = "Z" THEN
      700
510  IF W$ <  >  CHR$ (8) THEN 550
520  IF X$ = "" THEN  PRINT BL$;: GOTO 500
530  IF  LEN (X$) = 1 THEN X$ = "": PRINT BS$;: GOTO
      500
540 X$ =  LEFT$ (X$, LEN (X$) - 1): PRINT BS$;: GOTO
      500
550  IF W$ =  CHR$ (13) THEN  RETURN
560  PRINT BL$;: GOTO 500
700  PRINT W$;:X$ = X$ + W$: GOTO 500
```

1.4 ACCEPTING ONLY NUMBERS

When you are expecting a number and not a string, you want to reject letters and nonnumeric symbols.

In program ONLY NUMBER, line 500 tests for numbers instead of the whole (" " . . "^") range.

```
1  REM  ** ONLY NUMBERS **
20 BL$ =  CHR$ (7):BS$ =  CHR$ (8) + " " +  CHR$
     (8)
50  HOME : HTAB 3: VTAB 8: PRINT "AMOUNT:";: GOSUB
     450:V =  VAL (X$)
60  HTAB 3: VTAB 12: PRINT "COST:";: GOSUB 450:N =
      VAL (X$)
70  HOME : PRINT V,N: END
450 X$ = ""
500  GET W$: IF W$ >  = "0" AND W$ <  = "9" THEN
     700
510  IF W$ <  >  CHR$ (8) THEN 550
520  IF X$ = "" THEN  PRINT BL$;: GOTO 500
530  IF  LEN (X$) = 1 THEN X$ = "": PRINT BS$;: GOTO
     500
540 X$ =  LEFT$ (X$, LEN (X$) - 1): PRINT BS$;: GOTO
     500
550  IF W$ =  CHR$ (13) THEN  RETURN
560  PRINT BL$;: GOTO 500
700  PRINT W$;:X$ = X$ + W$: GOTO 500
```

1.5 CONTROLLING THE LENGTH

Another problem with the INPUT statement is that it does not check for any particular number of characters; you can type lines and lines of characters in response to it, and it will accept them patiently, until you reach 255, in which case a backslash (\) will be PRINTed, all the characters ignored, and a new INPUT line started.

Program LENGTH CONTROL behaves basically as the program RETURN does, except for lines 750 and 760. If the length of X$ (the characters that have been accepted) is equal to or greater than L (which must be passed as a parameter, so that input lines of different lengths can be processed by the same subroutine), the bell character is PRINTed and the character is rejected, even if it is valid. Otherwise, line 760 accepts it and adds it to X$.

```
1  REM  ** LENGTH CONTROL **
20 BL$ =  CHR$ (7):BS$ =  CHR$ (8) + " " +  CHR$
     (8)
50  HOME : HTAB 3: VTAB 8: PRINT "FIRST NAME (10
     CHAR.) ";:L = 10: GOSUB 450:N$ = X$
60  HTAB 3: VTAB 12: PRINT "MIDDLE INITIAL (1 CHA
     R.) ";:L = 1: GOSUB 450:I$ = X$
70  HOME : PRINT N$,I$: END
450 X$ = ""
```

```
500  GET W$: IF W$ >  = " " AND W$ <  = "Z" THEN
     750
510  IF W$ <  > CHR$ (8) THEN 550
520  IF X$ = "" THEN  PRINT BL$;: GOTO 500
530  IF  LEN (X$) = 1 THEN X$ = "": PRINT BS$;: GOTO
     500
540 X$ =  LEFT$ (X$, LEN (X$) - 1): PRINT BS$;: GOTO
     500
550  IF W$ =  CHR$ (13) THEN  RETURN
600  PRINT BL$;: GOTO 500
750  IF  LEN (X$) >  = L THEN  PRINT BL$;: GOTO 5
     00
760  PRINT W$;:X$ = X$ + W$: GOTO 500
```

1.6 ADDING A PROMPT LINE

It is very convenient to know exactly how many characters are expected in the INPUT.

Program PROMPT LINE initializes PC$ (prompt character) to the underscore, which is not available through the Apple keyboard; the space in BS$ is now replaced by an underscore.

Line 20 asks for your telephone number, L is set to 13 (to make room for the hyphens and parenthesis), and subroutine 450 is called.

Line 450 empties X$; if L is greater than 0, L underscores are PRINTed, and the cursor is returned to its original position PRINTing L backspaces, this time without erasing the underscore.

When the backspace is pressed, lines 520 through 540 behave as in the previous examples, except that when a character is erased it is replaced by an underscore.

When <return> is pressed line 553 erases the underscores after the last characters, before RETURNing.

Any other character can be used in the prompt line; the most commonly used are (besides the underscore) dots, hyphens, and slashes.

```
1  REM  ** PROMPT LINE **
10 PC$ =  CHR$ (95):BL$ =  CHR$ (7):BS$ =  CHR$ (
     8) + PC$ +  CHR$ (8)
20  HOME : HTAB 3: VTAB 12: PRINT "TELEPHONE:";:L
      = 13: GOSUB 450:T$ = X$
30  HTAB 1: VTAB 20: PRINT "YOUR TELEPHONE : ";T$
     : END
```

```
450 X$ = "": IF L > 0 THEN  FOR I = 1 TO L: PRINT
     PC$;: NEXT : FOR I = 1 TO L: PRINT  CHR$ (8)
     ;: NEXT
500  GET W$: IF W$ >  = " " AND W$ <  = "Z" THEN
     750
510  IF W$ <  >  CHR$ (8) THEN 550
520  IF X$ = "" THEN  PRINT BL$;: GOTO 500
530  IF  LEN (X$) = 1 THEN X$ = "": PRINT BS$;: GOTO
     500
540 X$ =  LEFT$ (X$, LEN (X$) - 1): PRINT BS$;: GOTO
     500
550  IF W$ <  >  CHR$ (13) THEN 600
553  IF  LEN (X$) < L THEN  FOR I =  LEN (X$) TO
     L - 1: PRINT " ";: NEXT
556  RETURN
600  PRINT BL$;: GOTO 500
750  IF  LEN (X$) >  = L THEN  PRINT BL$;: GOTO 5
     00
760  PRINT W$;:X$ = X$ + W$: GOTO 500
```

1.7 CONTINUING WHEN THE LENGTH IS REACHED

Program LENGTH CONTINUE incorporates a feature that is sometimes very useful: When the specified length is reached, the INPUT subroutine RETURNs, as if an automatic <return> had been typed.

Line 20 sets X to 2, and Y to 9; subroutine 1000 is called to send the cursor to locations X, Y. The message

"TODAY'S DATE (/ /)"

is PRINTed.

Line 30 sets X to 18 (Y has not changed) and subroutine 1000 is called once more. The length is set to two (two digits for the month), and the INPUT subroutine is called. With the cursor in the right place, the prompt line is placed in the month part of the date format created by line 20.

The INPUT subroutine only has one difference from the one in program PROMPT LINE. Line 760 echoes the accepted character, and if the length of X$ is equal to L, the subroutine RETURNs.

Line 30 accepts the month only if it is valid (1 - 12). Lines 40 and 50 set the cursor to the appropriate locations and behave as line 30. When line 60 is reached, the date is PRINTed using the individual parts: month, day, and year.

```
1  REM  ** LENGTH CONTINUE **
10 BL$ =  CHR$ (7):PC$ =  CHR$ (95):BS$ =  CHR$ (
    8) + PC$ +  CHR$ (8)
20  HOME :X = 2:Y = 9: GOSUB 1000: PRINT "TODAY'S
     DATE:  (  /  /  )"
30 X = 18: GOSUB 1000:L = 2: GOSUB 450:M =  VAL (
    X$): IF M < 1 OR M > 12 THEN  PRINT BL$;: GOTO
    30
40 X = 21: GOSUB 1000: GOSUB 450:D =  VAL (X$): IF
    D < 1 OR D > 31 THEN  PRINT BL$;: GOTO 40
50 X = 24: GOSUB 1000: GOSUB 450:A =  VAL (X$): IF
    A < 82 THEN  PRINT BL$;: GOTO 50
60  HOME :X = 8: GOSUB 1000: PRINT "TODAY'S DATE:
    "M"/"D"/"A: END
100  HOME : PRINT D$: PRINT A: END
450 X$ = "": IF L > 0 THEN  FOR I = 1 TO L: PRINT
    PC$;: NEXT : FOR I = 1 TO L: PRINT  CHR$ (8)
    ;: NEXT
500  GET W$: IF W$ >  = "0" AND W$ <  = "9" THEN
    760
510  IF W$ <  >  CHR$ (8) THEN 550
520  IF X$ = "" THEN  PRINT BL$;: GOTO 500
530  IF  LEN (X$) = 1 THEN X$ = "": PRINT BS$;: GOTO
    500
540 X$ =  LEFT$ (X$, LEN (X$) - 1): PRINT BS$;: GOTO
    500
550  IF W$ <  >  CHR$ (13) THEN 600
553  IF  LEN (X$) < L THEN  FOR I =  LEN (X$) TO
    L - 1: PRINT " ";: NEXT
556  RETURN
600  PRINT BL$;: GOTO 500
760  PRINT W$;:X$ = X$ + W$: IF  LEN (X$) = L THEN
     RETURN
770  GOTO 500
1000  HTAB X: VTAB Y: RETURN
```

1.8 DEFAULT VALUES

When entering a sequence of items of the same kind, sometimes you discover that many of the answers do not change every time, yet they must be retyped.

In program DEFAULT line 50 asks for a date; if you answer with a <return> and no other characters, the previous date is assigned to DA$. Otherwise the date PRINTed is the previous one.

The process is repeated for the receipt; finally line 80 PRINTs the final strings.

```
1  REM  ** DEFAULT **
10 BL$ =  CHR$ (7):PC$ =  CHR$ (95):BS$ =  CHR$ (
     8) + PC$ +  CHR$ (8)
20 PD$ = "12/03/82":PR = 500
30  HOME :X = 3:Y = 8: GOSUB 1000: PRINT "DATE (M
     M/DD/YY) ";:L = 8: GOSUB 450: IF X$ <  > "" THEN
     DA$ = X$:PD$ = DA$: GOTO 50
40 DA$ = PD$:X = 19: GOSUB 1000: PRINT DA$
50 X = 3:Y = 12: GOSUB 1000: PRINT "RECEIPT # (0-
     999) ";:L = 3: GOSUB 450: IF X$ <  > "" THEN
     R =  VAL (X$):PR = R: GOTO 80
60 R = PR:X = 21: GOSUB 1000: PRINT R
80  INVERSE :Y = Y + 2: GOSUB 1000: PRINT DA$:Y =
     Y + 2: GOSUB 1000: PRINT R: NORMAL : PRINT :
      END
450 X$ = "": IF L > 0 THEN  FOR I = 1 TO L: PRINT
     PC$;: NEXT : FOR I = 1 TO L: PRINT  CHR$ (8)
     ;: NEXT
500  GET W$: IF W$ >  = " " AND W$ <  = "Z" THEN
     760
510  IF W$ <  >  CHR$ (8) THEN 550
520  IF X$ = "" THEN  PRINT BL$;: GOTO 500
530  IF  LEN (X$) = 1 THEN X$ = "": PRINT BS$;: GOTO
     500
540 X$ =  LEFT$ (X$, LEN (X$) - 1): PRINT BS$;: GOTO
     500
550  IF W$ <  >  CHR$ (13) THEN 600
553  IF  LEN (X$) < L THEN  FOR I =  LEN (X$) TO
     L - 1: PRINT " ";: NEXT
556  RETURN
600  PRINT BL$;: GOTO 500
760  PRINT W$;:X$ = X$ + W$: IF  LEN (X$) = L THEN
      RETURN
770  GOTO 500
1000  HTAB X: VTAB Y: RETURN
```

1.9 DELETING THE ENTIRE LINE

It is very convenient, especially when entering long lines, to be able to erase the entire line when a mistake is discovered, instead of backspacing many times.

In Applesoft, if you want to delete or ignore a line that has been typed, you press ctrl-X.

Program DELETE LINE recognizes in lines 570 and 580 that a ctrl-X has been pressed: it then PRINTs as many BS$'s as characters are in X$, therefore returning to the same situation when the subroutine started.

```
1  REM  ** DELETE LINE **
10 BL$ =  CHR$ (7):PC$ =  CHR$ (95):BS$ =  CHR$ (
     8) + PC$ +  CHR$ (8)
20  HOME : HTAB 3: VTAB 9: PRINT "NAME : ";:L = 2
     0: GOSUB 450:N$ = X$
30  HOME : PRINT N$: END
450 X$ = "": IF L > 0 THEN  FOR I = 1 TO L: PRINT
     PC$;: NEXT : FOR I = 1 TO L: PRINT  CHR$ (8)
     ;: NEXT
500  GET W$: IF W$ >  = " " AND W$ <  = "Z" THEN
     760
510  IF W$ <  >  CHR$ (8) THEN 550
520  IF X$ = "" THEN  PRINT BL$;: GOTO 500
530  IF  LEN (X$) = 1 THEN X$ = "": PRINT BS$;: GOTO
     500
540 X$ =  LEFT$ (X$, LEN (X$) - 1): PRINT BS$;: GOTO
     500
550  IF W$ <  >  CHR$ (13) THEN 560
553  IF  LEN (X$) < L THEN  FOR I =  LEN (X$) TO
     L - 1: PRINT " ";: NEXT
556  RETURN
560  IF W$ <  >  CHR$ (21) THEN  PRINT BL$;: GOTO
     500
570  IF  LEN (X$) = 0 THEN  PRINT BL$;: GOTO 500
580  FOR J = 1 TO  LEN (X$): PRINT BS$;: NEXT : GOTO
     450
700  GOTO 500
760  IF  LEN (X$) = L THEN  PRINT BL$;: GOTO 500
770  PRINT W$;:X$ = X$ + W$: GOTO 500
```

1.10 RECUPERATING THE DELETED LINE

Program RECUPERATE LINE deletes the line with ctrl-X as in program DELETE LINE, but before doing so it stores X$ in TB$.

If ctrl-R is pressed anywhere in the INPUT subroutine, line 610 recognizes the case in which no characters are in X$, sets it to TB$, and jumps to line 625.

Line 620 PRINTs as many BS$'s as characters are in X$ (the prompt line is left as in the beginning of the INPUT subroutine), and sets X$ equal to TB$.

Line 625 PRINTs the new contents of X$. If its length is not equal to L, line 630 PRINTs as many PC$'s (underscores) as necessary to redraw the prompt line; PRINTing backspaces, the cursor returns to the location right after the last character in X$.

```
1  REM  ** RECUPERATE LINE **
10 PC$ =  CHR$ (95):BL$ =  CHR$ (7):BS$ =  CHR$ (
    8) + PC$ +  CHR$ (8)
15  HOME
20 X = 3:Y = 8: GOSUB 1000: PRINT "ITEM # ";:L =
    4: GOSUB 400:IT =  VAL (X$)
30 Y = 11: GOSUB 1000: PRINT "DESCRIPTION:";:L =
    20: GOSUB 400:D$ = X$
40 Y = Y + 2: GOSUB 1000: INVERSE : PRINT IT:Y =
    Y + 2: GOSUB 1000: PRINT D$: PRINT : NORMAL
    : END
400 TB$ = ""
450 X$ = "": IF L > 0 THEN  FOR I = 1 TO L: PRINT
    PC$;: NEXT : FOR I = 1 TO L: PRINT  CHR$ (8)
    ;: NEXT
500  GET W$: IF W$ >  = " " AND W$ <  = "Z" THEN
    750
510  IF W$ <  >  CHR$ (8) THEN 550
520  IF X$ = "" THEN  PRINT BL$;: GOTO 500
530  IF  LEN (X$) = 1 THEN X$ = "": PRINT BS$;: GOTO
    500
540 X$ =  LEFT$ (X$, LEN (X$) - 1): PRINT BS$;: GOTO
    500
550  IF W$ <  >  CHR$ (13) THEN 560
553  IF  LEN (X$) < L THEN  FOR I =  LEN (X$) TO
    L - 1: PRINT " ";: NEXT
556  RETURN
560  IF W$ <  >  CHR$ (21) THEN 600
570  IF X$ = "" THEN  PRINT BL$;: GOTO 500
580  FOR I = 1 TO  LEN (X$): PRINT BS$;: NEXT :TB
    $ = X$: GOTO 450
600  IF W$ <  >  CHR$ (18) THEN  PRINT BL$;: GOTO
    500
610  IF X$ = "" THEN X$ = TB$: GOTO 625
620  FOR I = 1 TO  LEN (X$): PRINT BS$;: NEXT :X$
     = TB$
625  PRINT X$;: IF  LEN (X$) = L THEN 500
630  FOR I = 1 TO L -  LEN (X$): PRINT PC$;: NEXT
    : FOR I = 1 TO L -  LEN (X$): PRINT  CHR$ (8
    );: NEXT : GOTO 500
750  IF  LEN (X$) >  = L THEN  PRINT BL$;: GOTO 5
    00
760  PRINT W$;:X$ = X$ + W$: GOTO 500
1000  HTAB X: VTAB Y: RETURN
```

1.11 RETURNING TO THE MENU

The <esc> key serves many purposes in Applesoft. It stands for "escape," and it is a good idea to use it to escape from an INPUT line to a different location.

When you have a menu and you choose one of the possible options, there should always be a way for you to return to it (if you press the wrong key) without having to go through the entire procedure of the option.

In program ESCAPE, line 600 recognizes <esc> and returns to the menu.

Use of the POP command erases one RETURN address since only 25 such addresses can be active simultaneously. In this case the subroutine is not RETURNing and that address would be kept and the number reduced needlessly.

The use of the boolean variable FL (for flag), in line 780, helps the input subroutine recognize if it must RETURN as soon as the length has been reached.

```
1  REM  ** ESCAPE **
2 BL$ =  CHR$ (7):PC$ =  CHR$ (95):BS$ =  CHR$ (8
     ) + PC$ +  CHR$ (8):TRUE = 1:FALSE = 0
10  HOME :X = 4:Y = 6: GOSUB 1000:FL = TRUE: PRINT
     "1-PERSONAL DATA":Y = Y + 4: GOSUB 1000: PRINT
     "2-VALUES"
20 Y = Y + 4: GOSUB 1000: PRINT "3-END":X = 6:Y =
     Y + 4: GOSUB 1000: PRINT "WHICH ? ";:L = 1
25 X = 16:Y = 18: GOSUB 1000: GOSUB 450:: IF X$ <
     "1" OR X$ > "3" THEN 25
30  ON  VAL (X$) GOTO 50,70,100
50 FL = FALSE: HOME :X = 3:Y = 8: GOSUB 1000: PRINT
     "NAME : ";:L = 20: GOSUB 450:N$ = X$: GOTO 1
     0
70 FL = FALSE: HOME :X = 3:Y = 9: GOSUB 1000: PRINT
     "VALUE : ";:L = 5: GOSUB 450:VL =  VAL (X$):
      GOTO 10
100  HOME :Y = 8:X = 3: GOSUB 1000: PRINT N$:Y =
     Y + 4: GOSUB 1000: PRINT VL: PRINT : END
450 X$ = "": IF L > 0 THEN  FOR I = 1 TO L: PRINT
     PC$;: NEXT : FOR I = 1 TO L: PRINT  CHR$ (8)
     ;: NEXT
500  GET W$: IF W$ >  = " " AND W$ <  = "Z" THEN
     760
510  IF W$ <  >  CHR$ (8) THEN 550
520  IF X$ = "" THEN  PRINT BL$;: GOTO 500
```

```
530   IF  LEN (X$) = 1 THEN X$ = "": PRINT BS$;: GOTO
      500
540 X$ =  LEFT$ (X$, LEN (X$) - 1): PRINT BS$;: GOTO
      500
550   IF W$ <  > CHR$ (13) THEN 560
553   IF  LEN (X$) < L THEN  FOR I =  LEN (X$) TO
      L - 1: PRINT " ";: NEXT
556   RETURN
560   IF W$ <  > CHR$ (21) THEN 600
570   IF  LEN (X$) = 0 THEN  PRINT BL$;: GOTO 500
580   FOR J = 1 TO  LEN (X$): PRINT BS$;: NEXT : GOTO
      450
600   IF W$ =  CHR$ (27) THEN  POP : GOTO 10
700   PRINT BL$;: GOTO 500
760   IF  LEN (X$) = L THEN  PRINT BL$;: GOTO 500
770   PRINT W$;:X$ = X$ + W$: IF  LEN (X$) < L THEN
      500
780   IF  NOT FL THEN 500
790   RETURN
1000  HTAB X: VTAB Y: RETURN
```

1.12 SELECTIVE INPUT

In the previous program, the use of <esc> allowed return to a menu, since there was only one. However, when the size of your program grows, so do the number of options and therefore the number of menus.

In program SELECTIVE, the main menu branches into two more menus, each having two options. The second menu should only accept numbers as input.

The variable FL will store the value of the previous menu (a value has been assigned to each level of menu), that is, the one to return to if <esc> is pressed while in the INPUT subroutine.

In line 10 it is assigned 0, because before the first menu there is no other one to return to. At this point, <esc> is interpreted as "end of the program."

Line 600 recognizes the <esc> and depending on the value of FL returns to menu #1, #2, or #3, or to the end of the program. The value of FL is adjusted in lines 10, 30, 70, and 200.

VL, a boolean variable, instructs the INPUT subroutine if it is expecting numbers or just any character.

F1, another boolean variable, makes the INPUT subroutine RETURN as soon as the length has been reached.

```
1  REM  ** SELECTIVE INPUT **
2 BL$ =  CHR$ (7):PC$ =  CHR$ (95):BS$ =  CHR$ (8
     ) + PC$ +  CHR$ (8):TRUE = 1:FALSE = 0
10 FL = 0: HOME :X = 3:Y = 8: GOSUB 5000: PRINT "
     1-PERSONAL DATA":Y = 12: GOSUB 5000: PRINT "
     2-VALUES"
20 X = 5:Y = 19: GOSUB 5000: PRINT "WHICH ? ";:L =
     1:F1 = TRUE: GOSUB 450: IF X$ <  > "1" AND X
     $ <  > "2" THEN  PRINT BL$;: GOTO 20
30 F1 = FALSE:FL = 1: ON  VAL (X$) GOTO 50,200
50 VL = FALSE:FL = 1: HOME :X = 3:Y = 6: GOSUB 50
     00: PRINT "1-OWNER":Y = 9: GOSUB 5000: PRINT
     "2-SPOUSE"
60 Y = 15:X = 5: GOSUB 5000: PRINT "WHICH ? ";:L =
     1:F1 = TRUE: GOSUB 450:V =  VAL (X$): IF V <
     1 OR V > 2 THEN  PRINT BL$;: GOTO 60
70 FL = 2:F1 = FALSE: ON V GOTO 80,100
80  HOME :X = 18:Y = 4: GOSUB 5000: PRINT "OWNER"
     :X = 6:Y = 12: GOSUB 5000: PRINT "NAME : ";:
     L = 20: GOSUB 450:OW$ = X$: GOTO 50
100 L = 20: HOME :X = 17:Y = 4: GOSUB 5000: PRINT
     "SPOUSE":Y = 14:X = 7: GOSUB 5000: PRINT "NA
     ME: ";: GOSUB 450:SN$ = X$: GOTO 50
150  HOME :X = 2:Y = 5: GOSUB 5000: PRINT "OWNER:
     "OW$:Y = 9: GOSUB 5000: PRINT "SPOUSE:"SN$
160 Y = 13: GOSUB 5000: PRINT "PRESENT VALUE:  "P
     V:Y = 17: GOSUB 5000: PRINT "PAST VALUE:  "O
     V: PRINT : END
200 FL = 1:F1 = TRUE:: HOME :X = 3:Y = 7: GOSUB 5
     000: PRINT "1-PRESENT":Y = 12: GOSUB 5000: PRINT
     "2-PAST"
210 X = 5:Y = 18: GOSUB 5000: PRINT "WHICH ? ";:L
      = 1: GOSUB 450:V =  VAL (X$): IF V < 1 OR V
      > 2 THEN  PRINT BL$;: GOTO 210
220 VL = TRUE:F1 = FALSE:FL = 3: ON V GOTO 240,30
     0
240  HOME :X = 14:Y = 4: GOSUB 5000: PRINT "PRESE
     NT VALUE":X = 14:Y = 14: GOSUB 5000: PRINT "
     $ ";
250 L = 9: GOSUB 450:PV =  VAL (X$): GOTO 200
300  HOME :X = 15:Y = 4: GOSUB 5000: PRINT "PAST
     VALUE":X = 14:Y = 14: GOSUB 5000: PRINT "$ "
     ;:L = 9
310  GOSUB 450:OV =  VAL (X$): GOTO 200
450 X$ = "": IF L > 0 THEN  FOR I = 1 TO L: PRINT
     PC$;: NEXT : FOR I = 1 TO L: PRINT  CHR$ (8)
     ;: NEXT
500  GET W$: IF W$ >  = " " AND W$ <  = "Z" THEN
     750
510  IF W$ <  >  CHR$ (8) THEN 550
520  IF X$ = "" THEN  PRINT BL$;: GOTO 500
```

```
530  IF  LEN (X$) = 1 THEN X$ = "": PRINT BS$;: GOTO
     500
540 X$ =  LEFT$ (X$, LEN (X$) - 1): PRINT BS$;: GOTO
     500
550  IF W$ <  > CHR$ (13) THEN 560
553  IF  LEN (X$) < L THEN  FOR I =  LEN (X$) TO
     L - 1: PRINT " ";: NEXT
556  RETURN
560  IF W$ <  > CHR$ (21) THEN 600
570  IF  LEN (X$) = 0 THEN  PRINT BL$;: GOTO 500
580  FOR J = 1 TO  LEN (X$): PRINT BS$;: NEXT : GOTO
     450
600  IF W$ =  CHR$ (27) THEN  POP : ON FL + 1 GOTO
     150,10,50,200
700  PRINT BL$;: GOTO 500
750  IF  NOT VL THEN 760
755  IF  VAL (W$) = 0 AND W$ <  > "0" THEN  PRINT
     BL$;: GOTO 500
760  IF  LEN (X$) = L THEN  PRINT BL$;: GOTO 500
765  PRINT W$;:X$ = X$ + W$: IF  LEN (X$) < L THEN
     500
770  IF  NOT F1 THEN 500
780  RETURN
5000  HTAB X: VTAB Y: RETURN
```

1.13 INPUT WITHOUT STOPPING THE PROGRAM

In all the previous cases, when some input was needed, the computer had to wait until you finished typing. This was signaled when either <return> was pressed or because the desired length was reached. In all cases, the program could not do anything but wait.

If the usual instructions for input (INPUT and GET) are used, all processing is interrupted until the input is finished.

One way to get input without interrupting the program in Applesoft is to read the paddles, but this is not very convenient for input of text.

Program NON-STOP uses the usual input subroutine, but instead of using GET to fetch individual characters, a new subroutine is called that reads the keyboard without stopping the program.

When a character is typed, the content of memory location –16384 becomes greater than 127. Therefore a program can know if a character is waiting to be read. POKE −16368, 0 signals that the character has been received and lowers the value of location −16384 to the real ASCII code of the character received.

Then, using the CHR$ ( ) function, the number in position −16384 is converted into the character typed.

If a key is pressed, subroutine 1000 returns its corresponding character in W$; otherwise, W$ returns empty.

The variable C is used as a counter to keep track of the cycles that have passed since the last time a key was pressed. When it becomes greater than 200, a bell character is PRINTed, and C is reset to 180; if 20 more cycles pass without any response from the keyboard, another bell is PRINTed. That way, when the person at the keyboard remains inactive for a certain amount of time, the computer gives him a reminder.

Notice that in line 560 we usually had a bell indicating that the character typed was not valid (like a control character other than the ones used the the INPUT subroutine). We cannot use this feature here because the program will pass through this instruction every time it finds an empty W$, which will be the case more often than not, and the bell would beep constantly.

Also, notice the absence of the blinking cursor. This happens because we are not using either GET or INPUT which turn it ON. If a blinking cursor is needed two things can be done: First, have a counter that causes an INVERSE blank to appear and disappear every time a certain value is reached; second, to PRINT a FLASHing space, which will blink by itself.

```
1  REM  ** NON-STOP **
2 BL$ =  CHR$ (7):PC$ =  CHR$ (95):BS$ =  CHR$ (8
    ) + PC$ +  CHR$ (8)
10  HOME : HTAB 3: VTAB 11: PRINT "NAME : ";:L =
    20: GOSUB 450:N$ = X$
20  HOME : PRINT N$: END
450 X$ = "": IF L > 0 THEN  FOR I = 1 TO L: PRINT
    PC$;: NEXT : FOR I = 1 TO L: PRINT  CHR$ (8)
    ;: NEXT
500  GOSUB 1000: IF W$ > = " " AND W$ < = "Z" THEN
    760
510  IF W$ < > CHR$ (8) THEN 550
520  IF X$ = "" THEN  PRINT BL$;: GOTO 500
530  IF  LEN (X$) = 1 THEN X$ = "": PRINT BS$;: GOTO
    500
540 X$ =  LEFT$ (X$, LEN (X$) - 1): PRINT BS$;: GOTO
    500
```

```
550   IF W$ <  >  CHR$ (13) THEN 560
553   IF  LEN (X$) < L THEN  FOR I =  LEN (X$) TO
      L - 1: PRINT " ";: NEXT
556   RETURN
560   IF W$ <  >  CHR$ (21) THEN 500
570   IF  LEN (X$) = 0 THEN 500
580   FOR J = 1 TO  LEN (X$): PRINT BS$;: NEXT : GOTO
      450
700   GOTO 500
760   IF  LEN (X$) = L THEN  PRINT BL$;: GOTO 500
770   PRINT W$;:X$ = X$ + W$: GOTO 500
1000   IF  PEEK ( - 16384) < 128 THEN W$ = "": GOTO
      1100
1010  W$ =  CHR$ ( PEEK ( - 16384) - 128): POKE  -
      16368,0:C = 0
1100  C = C + 1: IF C > 200 THEN  PRINT BL$;:C = 1
      80
1110   RETURN
```

1.14 INPUT WHILE PROGRAM IS BUSY

The INPUT subroutine of program INPUT WHILE BUSY is very similar to the one in program NON-STOP, but the C counter has been eliminated.

Instead of returning the character in W$, it is added to the previous contents of BF$ (a temporary buffer). That way, if the program is busy doing something else, that character is not lost and a new one can be read.

The program needs a file, TEST, in order to read from it and do input at the same time. The following line will create the file TEST:

```
10 D$ = CHR$ (4): ?D$ "OPEN TEST": ?D$ "WRITE TEST":
FOR I = 1 TO 200: ?1234567: NEXT: ?D$ "CLOSE"
```

Every pass through the loop calls subroutine 1000, which stores in BF$ a character, if one is waiting at the keyboard. In BF$ will be stored every character including control characters, escapes, and <return>.

When the program is done with the reading, it goes to the next instruction, which in turn calls the regular INPUT subroutine.

In this new version, instead of GETting a character, line 500 checks BF$; if it is empty it goes back to line 500. If it has exactly one character it copies it in W$ and empties BF$; W$ is processed the usual way. This case of only one character in BF$ must be considered separately to avoid the RIGHT$ (BF$, 0), which is illegal.

If BF$ has more than one character, W$ gets the leftmost one and deletes it from BF$ (BF$ = RIGHT$ (B$, LEN (BF$)–1)); this W$ is processed the usual way.

This way, not only regular input can be done while the program is busy, but also processes that normally must be completed before anything else can be done can be interrupted if needed. This applies to printer and disk operations which usually cannot be stopped except by RESETing, which should be considered a mistake.

In situations in which permanent input is needed and there are loops or repetitive structures of some kind, subroutine 1000 can be called every iteration to save the characters.

```
1  REM  ** INPUT WHILE BUSY **
2 BL$ =  CHR$ (7):BS$ =  CHR$ (8) +  CHR$ (95) +
     CHR$ (8):D$ =  CHR$ (4)
20  HOME :X = 16:Y = 6: GOSUB 2000: PRINT "READIN
    G":Y = 10:X = 10: GOSUB 2000: PRINT "YOU CAN
     TYPE SLOWLY"
25 Y = 14:X = 13: GOSUB 2000: PRINT "CONDITIONS:"
    :Y = Y + 2: GOSUB 2000: PRINT "ORDERS:":Y =
    Y + 2: GOSUB 2000: PRINT "METHODS:":Y = Y +
    2: GOSUB 2000: PRINT "DOCUMENT:"
30  PRINT D$"OPEN TEST,L15": GOSUB 1000: FOR I =
    1 TO 100: GOSUB 1000
35  PRINT D$"READ TEST,R"R:R = R + 20: IF R > 21 THEN
    R = 0
37  INPUT Q$: NEXT : PRINT D$"CLOSE"
40  HOME :X = 3:Y = 6: GOSUB 2000: PRINT "CONDITI
    ONS:";:L = 20: GOSUB 450:CO$ = X$
50 Y = 10: GOSUB 2000: PRINT "ORDERS :";:L = 15: GOSUB
    450:OD$ = X$
60 Y = 14: GOSUB 2000: PRINT "METHODS   :";:L = 2
    0: GOSUB 450:ME$ = X$
70 Y = 18: GOSUB 2000: PRINT "DOCUMENT : ";:L = 2
    0: GOSUB 450:DO$ = X$
80  HOME : PRINT : PRINT : PRINT "CONDITIONS:";CO
    $: PRINT : PRINT "ORDERS:";OD$: PRINT : PRINT
    "METHODS:";ME$: PRINT : PRINT "DOCUMENT:";DO
    $
90  END
450 X$ = "": FOR I = 1 TO L: PRINT  CHR$ (95);: NEXT
    : FOR I = 1 TO L: PRINT  CHR$ (8);: NEXT
```

```
500 W$ = "": GOSUB 1000: IF  LEN (BF$) = 0 THEN 5
     00
501  IF  LEN (BF$) = 1 THEN W$ = BF$:BF$ = "": GOTO
     505
503 W$ =  LEFT$ (BF$,1):BF$ =  RIGHT$ (BF$, LEN (
     BF$) - 1)
505  IF W$ >  = " " AND W$ <  = "Z" THEN 760
510  IF W$ <  >  CHR$ (8) THEN 550
520  IF X$ = "" THEN  PRINT BL$;: GOTO 500
530  IF  LEN (X$) = 1 THEN X$ = "": PRINT BS$;: GOTO
     500
540 X$ =  LEFT$ (X$, LEN (X$) - 1): PRINT BS$;: GOTO
     500
550  IF W$ <  >  CHR$ (13) THEN 560
553  IF  LEN (X$) < L THEN  FOR I =  LEN (X$) TO
     L - 1: PRINT " ";: NEXT
556  RETURN
560  IF W$ <  >  CHR$ (21) THEN 500
570  IF  LEN (X$) = 0 THEN  PRINT BL$;: GOTO 500
580  FOR J = 1 TO  LEN (X$): PRINT BS$;: NEXT : GOTO
     450
700  GOTO 500
760  IF  LEN (X$) = L THEN  PRINT BL$;: GOTO 500
770  PRINT W$;:X$ = X$ + W$: GOTO 500
1000  IF  PEEK ( - 16384) < 128 THEN  RETURN
1010 BF$ = BF$ +  CHR$ ( PEEK ( - 16384) - 128): POKE
      - 16368,0: RETURN
2000  HTAB X: VTAB Y: RETURN
```

1.15 CLEARING THE KEYBOARD

If you are using a GET statement to fetch characters, it is a good idea to erase any character that might have been typed previously in order to accept only those typed after the prompt line corresponding to the GET is PRINTed.

    POKE -16368, 0

clears the keyboard.

2

OUTPUT

2.1 FORMATTED OUTPUT

For the production of clear, understandable reports, either on the screen or on paper (and eventually on disk), it is necessary to control the length and position of every number and string PRINTed.

With constants there is no problem, since you can know exactly how long they are (e.g., PRINT 345.1 or PRINT "THE SUN IS SHINING"). With variables, it is another story: Strings can have from 0 to 255 characters and numbers from 1 (e.g., 3, 8, 1) to 15 (e.g., −6.95100521E+18).

For example: Program NO FORMAT asks for, and then PRINTS, the names and prices of spare parts needed to repair a computer. You want to produce a neat report for the customer.

Table 2.1 shows how it would look.

```
PART NAME : I.C.54717 <return>
PRICE : .95 <return>
PART NAME : CABLE I45-11 <return>
PRICE : 14.5 <return>
PART NAME : CPU <return>
PRICE : 380.55 <return>
PART NAME : <return>

              REPORT:
    PART NAME          PRICE
    ---------          -----
    I.C.54717            .95
    CABLE I45-11            14.5
    CPU         380.55
```

Table 2.1. Report without format.

As you can see, the prices are not aligned and the result looks rather messy. Imagine how it would look if you printed a combination of more than two items. In the last columns you wouldn't be able to tell what corresponded to what.

```
1  REM  ** NO FORMAT **
10  DIM N$(20),P(20): HOME
30  FOR I = 1 TO 20
40  INPUT "PART NAME :";N$(I): IF N$(I) = "" THEN
     N = I - 1: GOTO 60
50  INPUT "PRICE :";P(I): NEXT
60  PRINT : PRINT "             REPORT:"
70  PRINT "PART NAME                PRICE"
80  PRINT "=========                ====="
90  FOR I = 1 TO N: PRINT N$(I);"               "
     ;P(I)
100  NEXT
```

2.2 FORMATTING STRINGS

The easiest way to make sure a string's length is equal to a number you want is to use the LEFT$ function.

If, instead of PRINTing the string X$ as it is, you do:

PRINT LEFT$ (X$, 20);

you can be absolutely certain that no more than 20 characters will be PRINTed.

If X$ = "HELLO" and you do a PRINT LEFT$ (X$, 20), the computer will print HELLO and nothing else. Therefore, you must make sure the string has at least 20 characters before using LEFT$.

That is easy to do using the concatenation operator ("+"). In the previous example, it would look like:

PRINT LEFT$ (X$+"   ", 20);

This way, you can be sure X$ plus 20 blanks has at least 20 characters, and the length will always be the same.

However, there is an extra problem: If X$ has 236 or more characters before using '+', a 'STRING TOO LONG ERROR' will make the program stop. Since you now can control the input, you can make sure it has a certain amount of characters and not more; less maybe, but never more.

To make things easy, you can write a subroutine that PRINTs every string with the right length, like this:

```
5000 PRINT LEFT$ (X$+"                    ", L);:RETURN
```

That way, you send in X$ whatever you want to PRINT, and in L the desired length.

Notice that the function LEFT$ was used and not RIGHT$, since names and words are always PRINTed left justified.

2.3 FORMATTING NUMBERS

The differences between the formats of numbers and strings are unfortunately many more than the left or right justification. Numbers can come in a whole variety of forms: e.g., integers, as real numbers with a fractional part of unpredictable length, with or without a sign, in exponential form, etc. Thus, we must treat every case separately.

2.4 FIXING THE LENGTH

Before doing any formatting, you must decide how many decimal places you want when you PRINT the number. Otherwise, you will have the same problem of the strings. Let's suppose you are only interested in the integer part. The easy way would be:

```
PRINT RIGHT$ ("         "+STR$ (INT (X)), 9);
```

The INT function gets rid of the fractional part, STR$ makes it become a string, some spaces are added to the left to make sure it has the desired length, and finally it is right justified by RIGHT$ (this is the usual way with numbers).

However, when the number is negative, INT returns the smaller integer less than or equal to your X. When X has some fractional part, the computer will PRINT a number which is one less than what you expected. Fortunately, that is easy to solve:

```
100 IF X < 0 THEN FL = TRUE : X = ABS (X)
110 X$ = STR$ (INT (X))
120 IF FL THEN X$ = "–" + X$
130 PRINT RIGHT$ ("         " + X$, 9);
```

TRUE has been assigned 1 (boolean TRUE), and FL, a boolean variable, is set to TRUE. Later, when X has become a string with the correct value, we add the sign if necessary (a '+' sign could be added, too).

Integers are no problem, but fractions are a bit harder.

Subroutine 100 of program ELEMENTARY FORMAT returns in X$ the string equivalent to the value of N, right justified in a field of nine characters; only two (and always two) decimal places of N are returned.

Line 100 takes care of the case of a negative N; in line 110, X$ gets the integer part of N, a period, and the fractional part, which is produced by multiplying N by 100 and getting rid of the extra numbers using the INT function. Then N is converted into a string adding a couple of 0's (when N is less than 0.1, the result would have only one decimal number and the position of the period would be shifted, destroying the whole format).

If not two, but three decimal places are needed, every operation done with 100 is made with 1000, and three characters are used after the dot instead of two.

There is a small problem. When the integer part of the fractional part of N is evaluated, sometimes the result is not exact, and truncating it might result in the loss of some accuracy (try numbers like 0.7, 11.9, 3.3).

```
1  REM  ** ELEMENTARY FORMAT **
10  HOME :FALSE = 0:TRUE = 1
15  PRINT : INPUT "NUMBER:";N: GOSUB 100
20  PRINT : PRINT  RIGHT$ ("         " + X$,9): GOTO
     15
100 FL = FALSE: IF N < 0 THEN FL = TRUE:N =  ABS
     (N)
110 X$ =  STR$ ( INT (N)) + "." +  RIGHT$ ("00" +
      STR$ ( INT (N * 100)),2)
120  IF FL THEN X$ = "-" + X$
130 X$ =  RIGHT$ ("        " + X$,9): RETURN
```

2.5 ROUNDING

If we take a value like 1.299 and add .001 to it, the result is 1.300. Since the inaccuracy will not be greater than that (.001), the problem could be solved that way, before taking the integer part.

What would happen if the number did not have that problem to begin with? Let's see: if N = 3.55123, for example, it would become 3.55223, which through the INT function would yield the correct answer. But how about when N = 3.12999?

In that case, N would become 3.13. What we have done is to round up N when its third decimal digit is 9. This is usually considered correct and certainly solves our problem of the last section (if you did not want to round up those numbers that had a 9 in the third decimal place before the INT function might put it there, you could scan the STR$ of N, find the period, and see if there is a 9 or not. This becomes tedious, slow, and impractical).

If you are interested in keeping the precision to more decimal digits than this procedure allows, see program EXTENDED PRECISION, Section 11.15.

This method of rounding up 9's can be extended to any digit. It is very common to round numbers up that are over 5 in the digit that is going to be lost, and down if it is less than 5. To do this (in the case of two decimal digits), just add 0.005 to the number, multiply it by 100, get the INT part, and divide by 100. Here are two examples:

12.873 + .005 = 12.878
12.878 * 100 = 1287.8
INT (1287.8) = 1287
1287 / 100 = 12.87

48.178 + .005 = 48.183
48.183 * 100 = 4818.3
INT (4818.3) = 4818
4818 / 100 = 48.18

Program ROUND is a copy of program ELEMENTARY FORMAT with a rounding instruction in line 105.

In these two sections we have learned how to format numbers with two decimal places. Treating similar cases should be easy. To get three decimals, multiply by 1000 and add "000" to the STR$; to round the number, instead of adding .005 (or .001, whichever the case be), add .0005 (or .0001), etc., for any number of decimals.

```
1  REM  ** ROUND **
10  HOME :FALSE = 0:TRUE = 1
15  PRINT : INPUT "NUMBER:";N: GOSUB 100
20  PRINT : PRINT  RIGHT$ ("         " + X$,9): GOTO
    15
100 FL = FALSE: IF N < 0 THEN FL = TRUE:N =  ABS
    (N)
105 N = 100 * (N + 0.005) / 100
110 X$ =  STR$ ( INT (N)) + "." +  RIGHT$ ("00" +
     STR$ ( INT (N * 100)),2)
120  IF FL THEN X$ = "-" + X$
130 X$ =  RIGHT$ ("        " + X$,9): RETURN
```

2.6 FORMAT WITH SPECIAL SYMBOLS

In the previous section we added two decimal places and a dot to separate the integer part from the fractional. Now, we will add commas if the number is greater than 999 and 999999.

In program FORMAT WITH SYMBOLS, lines 100 and 110 do exactly the same job of program ROUND; line 120 looks for a dot.

In line 140 we already know how many digits we have remaining to the left of the dot (or how many digits there are, if we are dealing with an integer number). If there are less than 5, we do not have to worry about this number, and we send it to line 180; otherwise, in line 150 we insert a comma to mark the place of the thousands. The variable I contains now the number of digits remaining to the left. We repeat the process, and add another comma, if needed, to indicate that there are millions in this number.

Line 180 takes care of the minus sign and line 190 right justifies X$ to the desired length.

If we wanted the number to have a dollar sign to the left, we could insert:

175 X$ = ″$″ + X$

This way, the minus sign is placed before the dollar sign.

Changing this format is very easy now. Consider the following variations: We are using the number formatted by subroutine 100 in a report where a line of dots is being PRINTed between the description of the item and its price.

DOUGHNUTS $0.30
APPLE PIES $2.55
CHICKEN CURRY $11.00

Since we are adding spaces to the left of X$ just before setting the length (line 190), the result would be:

DOUGHNUTS...... $0.30

so, line 190 should be changed to

190 X$ = RIGHT$ (".................."+X$, 14) : RETURN

If we want to include a plus or minus sign at the end of the number, as in:

$1,500.00– $45,000.00+

line 180 should be changed to

180 IF FL THEN X$ = X$ + "–": GOTO 190

and a new line should be added:

185 X$ = X$ + "+"

```
1  REM  ** FORMAT WITH SYMBOLS **
2 FALSE = 0:TRUE = 1
10  HOME
20  PRINT : INPUT "NUMBER:";N: GOSUB 100
30  PRINT : PRINT X$: GOTO 20
100 FL = FALSE: IF N < 0 THEN FL = TRUE:N =  ABS
     (N)
110 X$ =  STR$ ( INT (N)) + "." +  RIGHT$ ("00" +
      STR$ ( INT ((N + .001) * 100)),2)
120  FOR I =  LEN (X$) TO 1 STEP  - 1: IF  MID$ (
     X$,I,1) = "." THEN 140
130  NEXT :I =  LEN (X$) + 1
140  IF I < 5 THEN 180
145 I = I - 3
150 X$ =  LEFT$ (X$,I - 1) + "," +  RIGHT$ (X$, LEN
     (X$) - I + 1)
160  IF I < 5 THEN 180
170 I = I - 3:X$ =  LEFT$ (X$,I - 1) + "," +  RIGHT$
     (X$, LEN (X$) - I + 1)
180  IF FL THEN X$ = "-" + X$
190 X$ =  RIGHT$ ("           " + X$,14): RETURN
```

2.7 FORMATTING VERY LARGE AND SMALL NUMBERS

In the previous programs of this section, when the number to be formatted came in exponential notation (e.g., 1.234E+23), the result was completely wrong, because a plus or minus sign or an ''E'' in the middle of the number was not expected. The case of the exponential notation has to be treated separately.

Program LARGE NUMBERS formats these numbers, but does nothing when N contains a number in standard form.

Since exponential form covers very large and very small numbers, rather than testing for values to see if it actually is in this notation, line 105 looks for an E in the third position of X$ (the STR$ of the number), from the left. But if the number does not have three digits, trying to evaluate MID$ (X$, LEN (X$)–3, 1) will cause an error; that is why in line 103 we check if N actually has at least three digits. If N does not have three digits or it does not have an E where needed, we return in X$ the number as it was before.

In line 110 we create a new string, S$, that stores the mantissa (the actual value without the exponent). In line 120 T gets the value of the exponent and in line 130 V is assigned the value of S$. Since the mantissa always comes in the same format (3.123234, 8.00452), we multiply it by 100000000 to get its real value.

Line 140 divides the two cases of numbers with positive or negative exponents. The case of a positive one is treated beginning in line 150: T (the exponent) is set to its real value (since after multiplying the mantissa by 10E+8 it no longer reflects the new value, and is high by 10E+8), we add to X$ (that now contains the mantissa without decimal numbers) as many 0's as the value of T.

Let's see an example:

```
N = 7.0095012E+15
S$ = 7.0095012                (S$=LEFT$ (X$, LEN (X$)–4))
T = 15                        (T=VAL (RIGHT$ (X$, 3)))
V = 700950120                 (V=VAL (S$)*100000000)
X$ = ''700950120''            (X$=STR$ (V))
T = 7                         (T=T–8)
X$ = ''7009501200000000''     (FOR I=1 TO T:X$=X$+''0'':NEXT)
```

Next, we send X$ to line 205, where a minus sign is added if N is negative, and we are done.

If N had a negative exponent, line 170 sets Y$ to "0.", taking care of one of the 0's in T. After that, 0's are added until the value of T−1 is reached. Then, the mantissa is added. To see the purpose of lines 180 and 190, let's consider the case in which N = 2.1E−11. The result we get in X$ is "0.00000000002100000000", which is numerically correct, but which has too many 0's at the end. Line 180 looks for the first nonzero number starting from the right and adjusts X$ so that it does not have all the 0's.

Line 205 takes care of the minus sign.

```
1  REM  ** LARGE NUMBERS **
2  HOME
5 TRUE = 1:FALSE = 0
10  INPUT "";W$:N =  VAL (W$)
20  GOSUB 100: PRINT X$: GOTO 10
100 NN = N:FL = FALSE: IF N < 0 THEN FL = TRUE:NN
     = ABS (N)
103 X$ =  STR$ (NN): IF  LEN (X$) <  = 3 THEN X$ =
     STR$ (N): RETURN
105  IF  MID$ (X$, LEN (X$) - 3,1) <  > "E" THEN
     X$ =  STR$ (N): RETURN
110 S$ =  LEFT$ (X$, LEN (X$) - 4)
120 T =  VAL ( RIGHT$ (X$,3))
130 V =  VAL (S$) * 100000000:X$ =  STR$ (V)
140  IF T < 0 THEN 165
150 T = T - 8: FOR I = 1 TO T:X$ = X$ + "0": NEXT

160  GOTO 205
165  REM
170 Y$ = "0.": FOR I = 1 TO  ABS (T) - 1:Y$ = Y$ +
     "0": NEXT :X$ = Y$ + X$
180 NN =  LEN (X$) + 1: FOR I =  LEN (X$) TO 1 STEP
     - 1: IF  MID$ (X$,I,1) <  > "0" THEN 200
190 NN = I: NEXT : GOTO 210
200 X$ =  LEFT$ (X$,NN - 1)
205  IF FL THEN X$ = "-" + X$
210  RETURN
```

2.8 OUTPUT WITH FIXED LENGTH

If X$ has too many characters when it is processed by

RIGHT$ ("   "+X$, L),

some of them are just chopped off, which can lead to confusing reports.

That is why it is a good idea to make note of that situation. One of the easiest ways to do this is not to return the chopped number in X$, but rather some sequence of symbols to indicate an irregular situation.

Program FIXED LENGTH is a copy of program FORMAT WITH SYMBOLS, with a small addition: In line 190, if the length of X$ is greater than L, X$ is filled with '>'s (which is easy to associate with "greater than"). Even though nothing abnormal is returned in terms of length or messages to the terminal, the person studying the report can see that something is wrong with that particular number.

```
1  REM  ** FIXED LENGTH **
10 TRUE = 1:FALSE = 0: HOME
20  PRINT : INPUT "NUMBER:";N:L = 10: GOSUB 100
30  PRINT : PRINT X$: GOTO 20
100 FL = FALSE: IF N < 0 THEN FL = TRUE:N =  ABS
     (N)
110 X$ =  STR$ ( INT (N)) + "." +  RIGHT$ ("00" +
      STR$ ( INT ((N + .001) * 100)),2)
120  FOR I =  LEN (X$) TO 1 STEP  - 1: IF  MID$ (
     X$,I,1) = "." THEN 140
130  NEXT :I =  LEN (X$) + 1
140  IF I < 5 THEN 180
145 I = I - 3
150 X$ =  LEFT$ (X$,I - 1) + "," +  RIGHT$ (X$, LEN
     (X$) - I + 1)
160  IF I < 5 THEN 180
170 I = I - 3:X$ =  LEFT$ (X$,I - 1) + "'" +  RIGHT$
     (X$, LEN (X$) - I + 1)
180 X$ = "$" + X$: IF FL THEN X$ = "-" + X$
190  IF  LEN (X$) > L THEN X$ = "": FOR I = 1 TO
     L:X$ = X$ + ">": NEXT : RETURN
200 X$ =  RIGHT$ ("          " + X$,L): RETURN
```

3

SORTING

One of the most frequent uses of the computer is to handle collections of objects such as the names and the numbers in a phone list, with the use of arrays.

Whenever one of those objects has to be changed, deleted, inserted, or handled in any way, it must be found in the array before anything can be done to it. Imagine looking for a name in a phone book that is not in alphabetical order!

Even though the computer is very fast, it is a waste of time to make it look for an element through an entire array, especially when there is a much more efficient way to do it.

There are several ways to sort objects in an array (the only way to keep collections of nonconstant objects in Applesoft). Some are faster than others, some longer, but no magic algorithm is faster, shorter, and more efficient than the rest.

Of the different methods discussed here, the Bubble sort is the easiest and the slowest; It is so short and easy that it is worth memorizing.

The fastest is the Shell sort, which I have found to be the most convenient for the size of collections dealt with in the Apple.

There are faster sorts (e.g., Heapsort, Quicksort), but their superiority becomes noticeable only when the number of items to sort is greater than 1000. Apart from being much larger programs than the Shell sort, they need an extra array of pointers, which, when dealing with more than 1000 items, greatly limits the memory availability.

I believe this version of the Shell sort is the most appropriate for a computer of the size of the Apple when working with Applesoft.

3.1 BUBBLE SORT

To understand easily the Bubble sort, let's first look at the way the smallest element in an array is found.

Program SMALLEST stores in M% ( ) a collection of numbers.

Line 100 assumes M% (1) is the smallest value of the array, and is going to compare it with every other element of M% ( ).

If the value in M% (I) is smaller than the one in M% (1), they are swapped, so that the smallest number found up to that point is in M% (1).

Notice that to do the swapping, a value must be stored in a temporary variable, since M% (1) = M% (J) destroys the value of M% (1).

At the end of the NEXT loop, the smallest element is in M% ( ).

```
1   REM  ** SMALLEST **
10   DIM M%(100)
20   FOR I = 1 TO 100:M%(I) =  INT (1000 *  RND (8
      )): NEXT
30   HOME : FOR I = 1 TO 100: PRINT M%(I)"-";: NEXT

40   PRINT
100   FOR I = 2 TO 100
110   IF M%(I) < M%(1) THEN T = M%(I):M%(I) = M%(1
      ):M%(1) = T
120   NEXT
130   PRINT "THE SMALLEST NUMBER IS: "M%(1)
```

In program BUBBLE SORT, line 10 enters all the values into M ( ); if we study lines 25 to 40 and forget temporarily about the rest, we see that it is exactly a search for the smallest, with the difference that instead of having chosen M (1) to be the smallest, we chose M (I) (which, at the start, is equal to M (1)).

Therefore, at the end of the first loop, the smallest number of the list is in M (1). I is incremented, and the J loop repeated, this time with J starting at I+1 (the second time, I+1 = 3). In other words, the J loop is looking for the smallest element of a smaller array, from 2 to N (since we already found the smallest of the entire list, and it is in M (1)). As I is incremented, smaller and smaller arrays are scanned for their smallest elements, and at the end, when I = N–1, only M (N–1) and M (N) are compared.

When the I loop terminates, all the elements are in ascending order.

```
1   REM  ** BUBBLE SORT **
5   HOME
10   VTAB 8: INPUT "NUMBER OF ITEMS TO SORT:";N: PRINT
      : DIM M(N): FOR I = 1 TO N: PRINT "M("I")=";
      : INPUT "";M(I): NEXT
20   FOR I = 1 TO N - 1
25   FOR J = I + 1 TO N
30   IF M(I) > M(J) THEN T = M(I):M(I) = M(J):M(J)
       = T
40   NEXT
45   NEXT
50   FOR I = 1 TO N: PRINT M(I): NEXT
```

3.2 DELAYED EXCHANGE BUBBLE SORT

Notice how in program BUBBLE SORT an element must be moved around many times until it finds its final position. All the swappings are not strictly necessary.

In program DELAYED EXCHANGE, line 20 sets M (the assumed smallest of the subarray from I to N) to the value of M (I), just like in the program SMALLEST. The index of M ( ) is stored in K; lines 25 through 40 have a J-loop similar to the one in the Bubble sort. The difference is that if it finds that M (J) is smaller than M, it stores M (J) in M (therefore M has the value of the smallest one found so far), and its index in K. This avoids all the swapping. When the J-loop is completed, one swapping is made: The smallest value found (M (K)) is exchanged with M (I).

The increase in speed is considerable.

```
1   REM  ** DELAYED EXCHANGE **
5   HOME
10   INPUT "NUMBER OF ITEMS TO SORT:";N: DIM M(N):
       FOR I = 1 TO N: PRINT "M("I;: INPUT ") = ";
      M(I): NEXT
20   FOR I = 1 TO N - 1:M = M(I):K = I
25   FOR J = I + 1 TO N
30   IF M > M(J) THEN M = M(J):K = J
40   NEXT
45 T = M(I):M(I) = M(K):M(K) = T: NEXT
50   FOR I = 1 TO N: PRINT M(I): NEXT
```

3.3 INSERTION SORT

When you have the elements in memory and you use either one of the previous Bubble sorts, the program stops any activity until the sort is finished. This is sometimes inconvenient or annoying for the impatient user.

A different way to do the sorting is to place every element in its corresponding location at the moment of input. At first, this sounds like the ideal solution, but it is inconvenient: When the element just entered must be placed in the middle or the beginning of the array, all the remaining elements must be moved one place to the right (or to the bottom, depending on how you visualize an array) to make room for the new one. If the incoming elements are nearly in ascending order, or if the collection is not too big, this does not cause any noticeable delay, but, when you are dealing with large arrays, it becomes very slow. Yet, most users will prefer a very short delay in the input rather than some minutes of apparent inactivity.

In program INSERTION SORT, line 10 prompts for the amount of numbers to sort, DIMensions M ( ), and starts the input. T stores every incoming number, and line 20 starts a loop that looks through the subarray (from 1 to I, because there are no more than I elements) until it finds a place where the previous number is smaller or equal to T, and the next one is bigger. If this happens before the end of the J-loop, line 40 moves every element after J one position, making room for T. Then, T is assigned to the place where it belongs in the ordered array.

If the J-loop terminates without finding a place for T, it means it is greater than every element of the array, and T is placed in M (I). This way, every new element in the list is put in its correct place. At the end of the input the array is sorted.

```
1  REM  ** INSERTION SORT **
10  HOME : VTAB 5: INPUT "NUMBER OF ITEMS TO SORT
     :";N: DIM M(N): FOR I = 1 TO N: INPUT "NEW N
     UMBER:";T
20  FOR J = 1 TO I: IF T < M(J) THEN 40
30  NEXT J:M(I) = T: GOTO 60
40  FOR K = I TO J STEP  - 1:M(K) = M(K - 1): NEXT

50 M(J) = T
60  FOR J = 1 TO I: PRINT M(J): NEXT
70  NEXT I
```

3.4 SHELL SORT

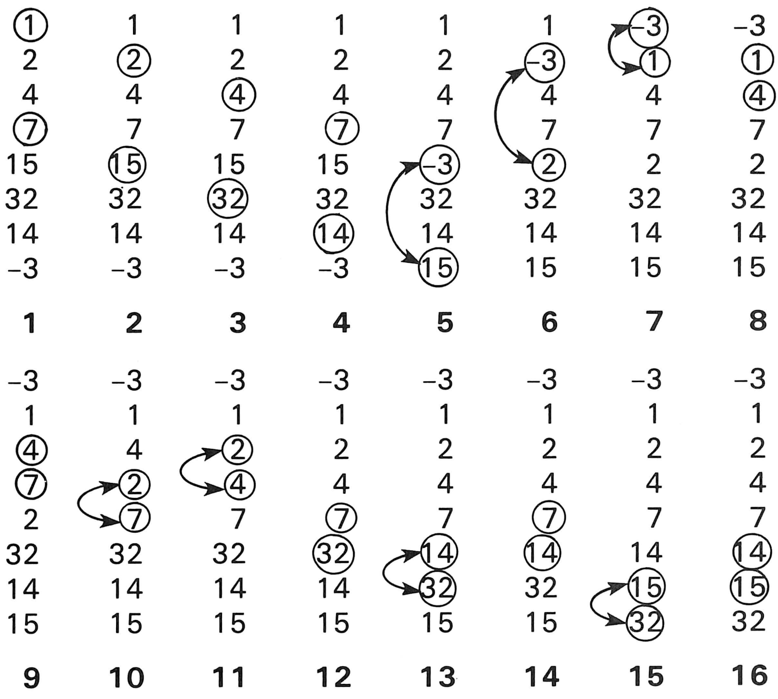
This is a remarkably fast sort and the actual program is very short.

Before studying the program, let's see an example:

We start with the situation shown in column 1 of Table 3.1. The SHELL SORT uses a guide (D in the program) to compare elements D positions apart. This guide is a power of 2 minus 1 (1, 3, 7, 15...) and the first one used is always the highest one which is smaller than the number of elements in the array.

In Table 3.1 the elements compared are circled, and the arrows indicate the swappings.

Table 3.1. Sorting with the SHELL SORT.



1	2	3	4	5	6	7	8
1	1	1	1	1	1	-3	-3
2	2	2	2	2	-3	1	1
4	4	4	4	4	4	4	4
7	7	7	7	7	7	7	7
15	15	15	15	-3	2	2	2
32	32	32	32	32	32	32	32
14	14	14	14	14	14	14	14
-3	-3	-3	-3	15	15	15	15

9	10	11	12	13	14	15	16
-3	-3	-3	-3	-3	-3	-3	-3
1	1	1	1	1	1	1	1
4	4	2	2	2	2	2	2
7	2	4	4	4	4	4	4
2	7	7	7	7	7	7	7
32	32	32	32	14	14	14	14
14	14	14	14	32	32	15	15
15	15	15	15	15	15	32	32

Starting with the guide 3, column 1 shows how elements 1 and 1+guide (4) are compared; since they do not have to be swapped, we go on to column 2.

The first swapping is performed in column 5; column 6 shows how –3 is made to sink as far as it can go.

Since in line 5 the highest element compared was already at the end of the array (the −3), line 7 starts from the bottom with the new guide, 1; 3 and 1 are swapped.

The next swapping takes place in column 10; column 11 shows how the 2 sank to its definite location. Columns 13 and 15 do swappings, and in line 16 the array is sorted.

This technique works every time and has been demonstrated mathematically. It can be found in nearly any book on data handling or data structures. The fact is, every time you add one element to an array to be sorted with any of the previous methods (BUBBLE, DELAYED EXCHANGE, INSERTION), the amount of comparisons can be as bad as the cube of the number of elements. In SHELL SORT the time grows nearly as a straight line.

Now let's study program SHELL SORT.

Lines 10 and 20 do the input of the elements of M ( ), and line 30 calls subroutine 1800, the Shell sort. The first thing it does is set D (the guide) to 1, and in line 1810 multiply it by 2 several times, until it becomes greater than N (the number of elements in the array).

Line 1820 subtracts 1 from D and divides it by 2 to find the power of 2 immediately below N. In order to show how far the sort is from completion, D is PRINTed on the screen. When D gets to 0, the sort is finished.

Line 1830 uses I to go through the array; J is originally set to I and K to J+D, so that elements J and K (which is D numbers above J) are compared. If they are swapped, J is decremented D units, and K is set to J+D, to make this swapped element sink to the bottom as far as possible.

When all the array has been scanned (I actually goes only until N–D to avoid attempts to compare elements outside the dimensioned M ( )), line 1820 finds the new guide number and the process is repeated.

```
1   REM  ** SHELL SORT **
10   HOME : VTAB 6: INPUT "NUMBER OF ITEMS TO SORT
      :";N: DIM M(N)
20   FOR I = 1 TO N: PRINT "M("I")=";: INPUT "";M(
      I): NEXT
30   GOSUB 1800: HOME : FOR I = 1 TO N: PRINT M(I)
      : NEXT : END
1800 D = 1
1810 D = 2 * D: IF D < N THEN 1810
```

```
1820 D =  INT ((D - 1) / 2): HTAB 23: VTAB 13: PRINT
     D" ";: IF D = 0 THEN  RETURN
1830  FOR I = 1 TO N - D:J = I
1840 K = J + D: IF M(K) < M(J) THEN T = M(J):M(J)
      = M(K):M(K) = T:J = J - D: IF J > 0 THEN 18
     40
1850  NEXT I: GOTO 1820
```

3.5 TAGFILES

So far, all the examples have dealt with numbers, yet many times strings are what has to be sorted. This does not change the actual procedure because the computer can compare strings the same way it can numbers; it can swap them and also do the same operations it does with numbers. But the treatment of strings in memory is not as neat in Applesoft (and in most BASICs) as that of numbers, and some problems may appear.

Not only does the handling of strings pose special difficulties when too many swappings must be made, but also there are cases in which the collection of items to be sorted does not consist of a single item but of related entries, as is the case when you are dealing with a person's name, address, telephone number, social security number, etc.

If you must swap the entire collection every time you find one item which must be moved (or else the final array will have names that do not correspond to the addresses, etc.), the process will become impractically long.

The Tagfile method will be more easily understood with an example.

Column A of Table 3.2 shows the inital situation of two arrays that are going to be sorted.

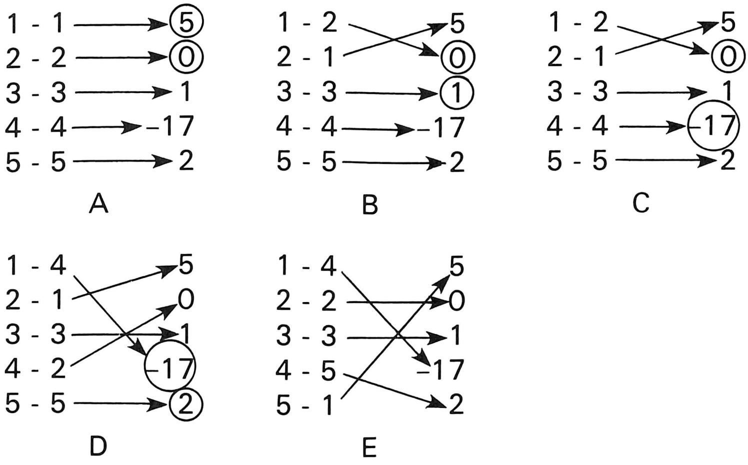
The left column corresponds to the indices, the second column is the array TAGFILE ( ), and the third column is the actual array to be sorted.

A Bubble sort will be used.

Notice that the tagfile has the numbers in a natural order at the beginning.

The two first elements are compared as indicated by the circles around them. They are found to be in the wrong order. However, instead of swapping the actual numbers in the array, column B shows how the elements of the tagfile were interchanged.

Table 3.2. BUBBLE SORT with TAGFILE.



Now the smallest element is pointed at by TAGFILE (1) and the actual array has not been touched.

The comparisons are not made using the indices I or J (as in the BUBBLE SORT program), but rather using TAGFILE (I) and TAGFILE (J).

Column C shows how 0 and −17 are compared and the way the tagfile was arranged in column D.

Column E shows the final situation; the rest of the sort is not shown, because by now a bubble sort should be clear. The important fact to understand is that the comparisons and the swappings are performed in the tagfile and not in the array.

This way, if the array to be sorted has more items related to it, the entire collection would still be in order after the sort because the only change is made in the tagfile.

In program TAGFILE, a collection of names, addresses, telephone numbers, and social security numbers will be stored and sorted in four arrays: N$ ( ), AD$ ( ), TE$ ( ), and S$ ( ). The actual tagfile will be in TF ( ).

Line 10 prompts for the number of names, to DIMension the arrays; line 20 sets all the elements of the tagfile equal to their index. Lines 30 through 70 do the input and lines 80 through 100 do a regular BUBBLE SORT. Notice that the only difference with a regular BUBBLE SORT is the comparisons (N$ (TF (I)) instead of

(N$ (I)), and the swappings:

T = TF (I) : TF (I) = TF (J) : TF (J) = T

Line 110 PRINTs the sorted array, using TF (I) as index, instead of I alone.

Even though we used a bubble sort to make the program shorter, a shell sort could have been used as well.

```
1   REM  ** TAGFILE **
10   HOME : VTAB 6: INPUT "NUMBER OF NAMES : ";N: DIM
      N$(N),AD$(N),TE$(N),SS$(N),TF(N)
20   FOR I = 1 TO N:TF(I) = I: NEXT
30   FOR I = 1 TO N: HOME : INPUT "NAME ";N$(I)
35   IF N$(I) = "" THEN 80
40   PRINT : INPUT "ADDRESS ";AD$(I)
50   PRINT : INPUT "TELEPHONE ";TE$(I)
60   PRINT : INPUT "SOCIAL SECURITY ";SS$(I)
70   NEXT
80   FOR I = 1 TO N - 1: FOR J = I + 1 TO N
90   IF N$(TF(I)) > N$(TF(J)) THEN T = TF(I):TF(I)
      = TF(J):TF(J) = T
100   NEXT : NEXT
110   FOR I = 1 TO N: PRINT N$(TF(I))" "AD$(TF(I))
      " "TE$(TF(I))" "SS$(TF(I)): NEXT
```

3.6 MULTIPLE TAGFILES

Since the original information is not changed in any way, we can have not only one tagfile, but also as many as we want, by using each one to keep track of the order of the arrays and using different parts of it as the ordering criteria. Program MULTIPLE TAGFILE is very similar to Program TAGFILE, except that three tagfiles are used.

In line 10, apart from the arrays used in the previous program, T1 ( ) and T2 ( ) are DIMensioned as TF ( ).

Line 20 sets the three tagfiles' elements to their index values, and lines 30 through 70 do the INPUT.

The Bubble sort of lines 80 through 100 actually does three sorts: One for TF ( ) using the name as the ordering criteria, one for T1 ( ) using the address, and another for T2 ( ), using the social security number. Line 105 asks which of the three orders is to be shown.

```
1  REM   ** MULTIPLE TAGFILE
10  HOME : VTAB 6: INPUT "NUMBER OF NAMES : ";N: DIM
     N$(N),AD$(N),TE$(N),SS$(N),TF(N),T1(N),T2(N)

20  FOR I = 1 TO N:TF(I) = I:T1(I) = I:T2(I) = I:
      NEXT
30  FOR I = 1 TO N: HOME : INPUT "NAME ";N$(I)
40  PRINT : INPUT "ADDRESS ";AD$(I)
50  PRINT : INPUT "TELEPHONE ";TE$(I)
60  PRINT : INPUT "SOCIAL SECURITY ";SS$(I)
70  NEXT
80  FOR I = 1 TO N - 1: FOR J = I + 1 TO N
90  IF N$(TF(I)) > N$(TF(J)) THEN T = TF(I):TF(I)
      = TF(J):TF(J) = T
93  IF AD$(T1(I)) > AD$(T1(J)) THEN T = T1(I):T1(
     I) = T1(J):T1(J) = T
96  IF SS$(T2(I)) > SS$(T2(J)) THEN T = T2(I):T2(
     I) = T2(J):T2(J) = T
100  NEXT : NEXT
105  HOME : PRINT "1-NAME": PRINT "2-ADDRESS": PRINT
     "3-SOCIAL SECURITY ";: GET W$: HOME : ON VAL
     (W$) GOTO 110,120,130
110  FOR I = 1 TO N: PRINT N$(TF(I))" "AD$(TF(I))
     " "TE$(TF(I))" "SS$(TF(I)): NEXT
112  GET W$: GOTO 105
120  FOR I = 1 TO N: PRINT AD$(T1(I))" "N$(T1(I))
     " "TE$(T1(I))" "SS$(T1(I)): NEXT
122  GET W$: GOTO 105
130  FOR I = 1 TO N: PRINT SS$(T2(I))" "N$(T2(I))
     " "TE$(T2(I))" "AD$(T2(I)): NEXT
140  GET W$: GOTO 105
```

3.7 BINARY SEARCH

When an array is sorted, it is very easy to look for a specific item, just the way we do when looking for a number in the phone book. We open it in the middle, and if the letter we find is bigger (alphabetically bigger), we look before that page; otherwise, we look after that page. The process is repeated until either we find it or we know it is not there.

The process is the same for a binary search. Let's see an example:

The array is:

1 - 67 - 68 - 71 - 89 - 102 - 345 - 356 - 367 - 456

and we want to find 367.

First, we see how many elements there are (10) and find the middle point (5).

We look at the fifth position. If we find the number, the operation is finished. Otherwise, we see if the one we found is bigger or smaller than 367. In this case 89 is smaller than 367 and we know that our number cannot be in the first half of the array.

We repeat the process considering only the subarray going from 102 to 456, or from positions 6 to 10. If we come to a point in which our subarray has only two elements and none is the one we are looking for, we can be certain 367 is not in the array; otherwise, we would have found it.

The new middle point is 8; we compare 356 with our 367 and deduce that we still have to try with a smaller array, this time from positions 9 to 10. The next comparison finds the number.

In program BINARY SEARCH, line 10 does the INPUT of the strings and subroutine 1800 (a Shell sort) sorts them.

Line 80 asks for the string to be found, then B is set to 1 and E to N. B will point to the beginning of the current subarray and E will point to the end. At the beginning, the subarray will be the entire M$ ( ).

MP is assigned the average of E and B (MP contains, then, the middle point of the subarray); line 110 sees if the element in the middle of the subarray is equal to AB$. If it is, the program jumps to line 200 and shows where AB$ is in the array. Otherwise, line 115 sees if MP is greater than B, which would mean we are dealing with a unit-long array. If this is the case, line 117 compares AB$ with the element in the array after M$ (MP), because when the average of B and E is found, there is the possibility of not considering the last element of the subarray, due to the fractional part lost with the INT function.

If that is the element, then MP = MP + 1 (which is where the number was found), and line 200 PRINTs the position of AB$ in the array; otherwise, the number is not in the array and line 150 tells us exactly that.

In line 120, if M$ (MP) is greater than AB$, then E = MP – 1, which means that the subarray is going to be the lower half of the original one, otherwise, list 130 sets B to MP + 1 and the new subarray is the upper half.

Since every time through the cycle the array is divided by two, we are eliminating a great number of comparisons.

```
1  REM  ** BINARY SEARCH **
5  HOME
10  HOME : VTAB 6: INPUT "NUMBER OF ITEMS TO SORT
     :";N: DIM M$(N): FOR I = 1 TO N: PRINT "M$("
     I")=";: INPUT "";M$(I): NEXT
20  GOSUB 1800: HOME : FOR I = 1 TO N: PRINT M$(I
     ): NEXT
80  PRINT : INPUT "ITEM TO FIND:";AB$: IF AB$ = "
     " THEN  HOME : END
90 B = 1:E = N
100 MP =  INT ((B + E) / 2)
110  IF M$(MP) = AB$ THEN 200
115  IF MP > B THEN 120
117  IF AB$ <  > M$(MP + 1) THEN 150
118 MP = MP + 1: GOTO 200
120  IF M$(MP) > AB$ THEN E = MP - 1: GOTO 100
130 B = MP + 1: GOTO 100
150  PRINT : PRINT "NOT FOUND": GOTO 80
200  PRINT : PRINT AB$" IS IN POSITION "MP: GOTO
     80
1800 D = 1: REM  ** SHELL SORT **
1810 D = 2 * D: IF D < N THEN 1810
1820 D =  INT ((D - 1) / 2): IF D = 0 THEN  RETURN

1830  FOR I = 1 TO N - D:J = I
1840 K = J + D: IF M$(K) < M$(J) THEN T$ = M$(J):
     M$(J) = M$(K):M$(K) = T$:J = J - D: IF J > 0
      THEN 1840
1850  NEXT I: GOTO 1820
```

3.8 POINTERS

A completely different sorting technique can be used if the elements of the array are considered to be parts of what is called a linked list.

Even though it is much slower than the Shell sort (but faster than the Bubble sort), when one element is eliminated from the array, the order is still preserved, while in all the previous sorts the process had to be started all over again. After each insertion and deletion, this sort preserves the order.

The best way to understand it is through an example.

Here is a list with three numbers: 13, 29, and 40. Figure 3.1 shows how each one has a pointer to the next element in the list.

A new element has been read, and must be added to the list. This number is 17. Instead of adding it randomly, and shifting every element of the list, Figure 3.2 shows how it is added so that the pointers are rearranged to follow the numerical order.

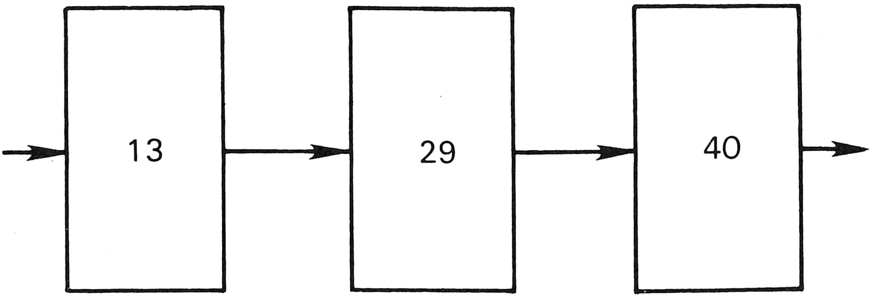


Figure 3.1. Linked list.

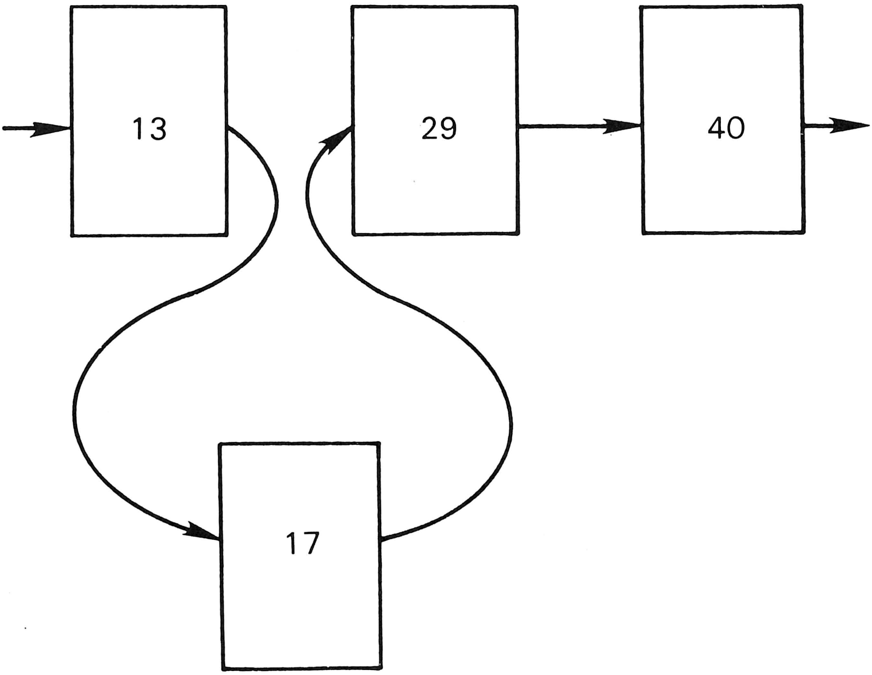


Figure 3.2. Addition of element to list.

Now let's transfer this idea into Applesoft. Figure 3.3 shows the original list.

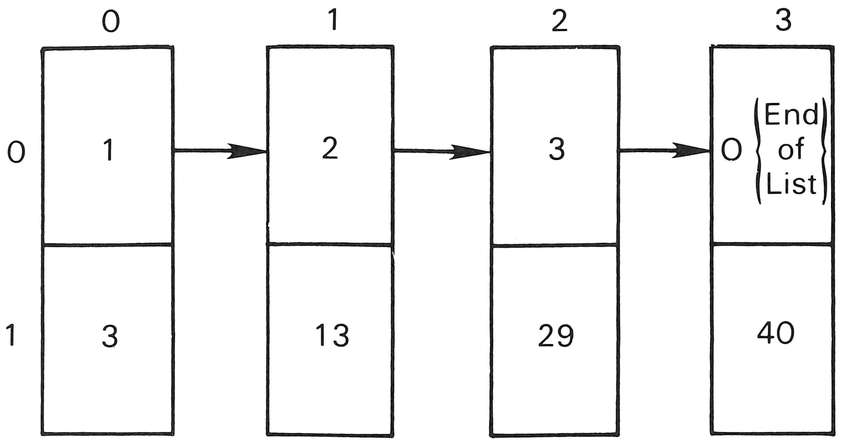


Figure 3.3. List in Applesoft.

The array was DIMensioned as M (900, 1), so that we have a collection of M (I, 0), and a collection of M (I, 1). The M (I, 0)'s will be the pointers while the M (I, 1)'s will store the actual values. The pointers will be the indices of M (I, X) where the next element will be located. In the example, M (0, 0) = 1 shows that the first element of the list is in position M (1, 0) and M (1, 1).

This element M (0, 0) will be called the head of the list, since it will be pointing at the first element even when the list is empty. Because this element with index 0 has been chosen to be different from the others and will never store any of the elements of the list, the number 0 can be used as a convention to indicate the end of the list. Therefore, when the list is empty, M (0, 0) = 0.

The number stored in M (0, 1) will be the current number of elements in the list.

In the example, all the elements are in order and therefore every M (I, 0) = I + 1. Figure 3.4 shows how the number 17 was added to the list.

To scan the list and find the right position for 17, we first look at the head, M (0, 0). If it is 0, we know that the list is empty, 17 must be stored in M (1, 1) and 1 in M (0, 0), since this is the first element of the list, and 0 must be stored in M (1, 0), indicating M (1, 1) is the last element of the list.

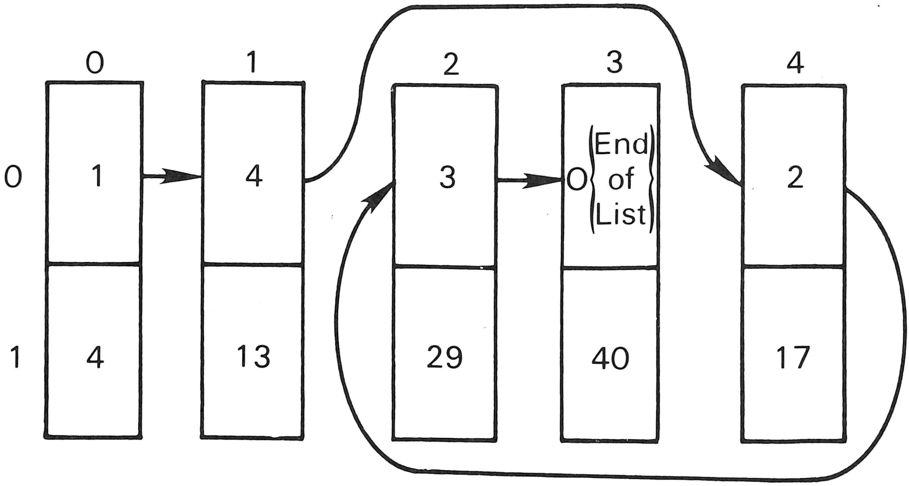


Figure 3.4. Addition of element to list in Applesoft.

M (0, 1) is used to keep track of how many elements are currently on the list, and in this case M (0, 1) = 1.

Since M (0, 0) is not 0, we know there is at least one element, because M (0, 0) = 1, we go ahead and look at M (1, 1). The element there is 13, but 17 is greater, so we still must continue the search. M (1, 0) = 2, and we see that M (2, 1) = 29, which is greater than 17. Therefore, 17 must be placed between M (1, 1) and M (2, 1), not necessarily physically, but we have to make the pointers indicate it is between the two.

Through M (0, 1) we know that three elements are on the list and the first empty place is M (4, X). We set M (4, 1) = 17, and proceed. The element immediately before 17 is M (1, 1); therefore, we remember its pointer (M (1, 0) = 2), and set M (1, 0) to the index of where 17 is now, 4. The element after 17 must be 19, which is where M (1, 0) was pointing. Therefore, we store the previous pointer of M (1, 0) (which we were careful to store somewhere) in M (4, 0), and the final situation is the one in Figure 3.6.

The procedure above is the same used to add any number, regardless of where in the list it is going to be, or how big the list is.

Deleting one element from the list is a bit more complicated. Starting with the situation in Figure 3.4, we decide to eliminate the 29. First we look for it, to see where it is (not in the array, but in the list: It is important to know which element was pointing at the 29 before deleting it). M (4, 0) is pointing to it and the 29 is pointing to M (3, X). The first step in the elimination is to change the pointers by simply making M (4, 0) point at the element at which the 29 was pointing. The 29 is completely out of the list.

The situation is shown in Figure 3.5.

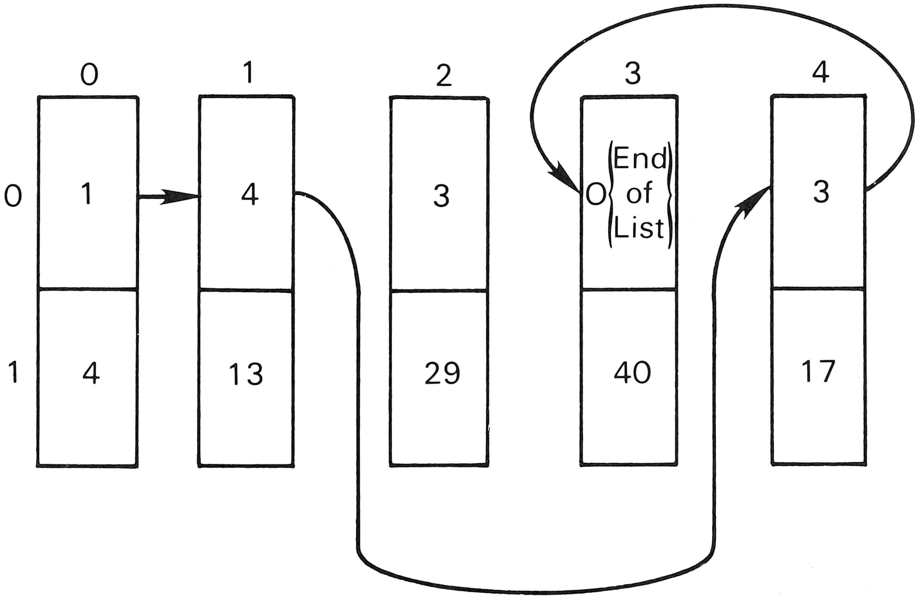


Figure 3.5. Elimination of element from list.

Now we have an empty place (where the 29 was, which now is not being pointed at by anything) which we should occupy. The most sensible thing to do is to move the last element and take care of its pointers instead of shifting every element one place to the left.

It is always important to know which element is pointing at the last one in the list. We know that 29 was in position 2, therefore we move M (4, 1) to M (2, 1). M (4, 0) was pointing at 3, therefore M (2, 0) = 3, and the element that was pointing at M (4, X), which was M (1, 0), is set to M (2, 0). This situation is shown in Figure 3.6.

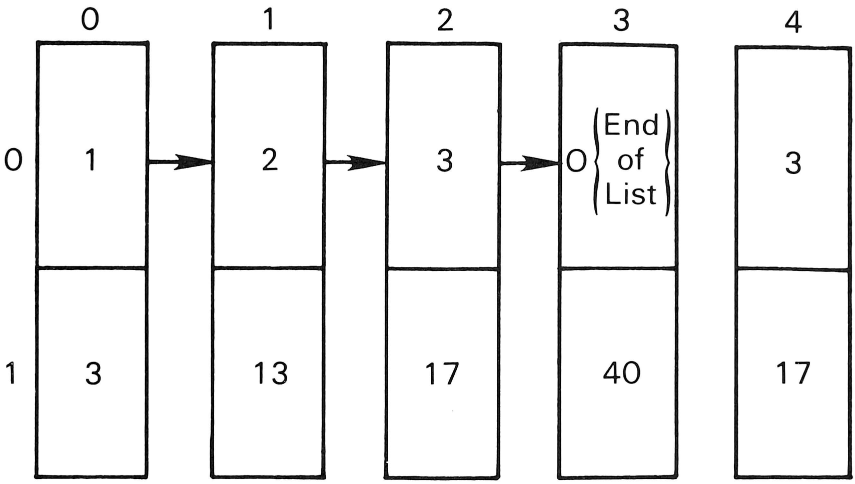


Figure 3.6. Final list.

If the list is very long, looking for an element can be a lengthy process. A binary search cannot be used because the list is sorted sequentially, and we do not know which is the middle element of the list until we read all the previous ones. I have devised what I call pseudo-binary search, which, if not as efficient as the real one, is much faster than the sequential search.

We look at the number of elements on the list (M (0, 1)), and then look every 20 elements (the increment can vary for very large or very small lists, but this is a good average) of the array, not of the list, to see if its pointer is pointing at an element just under the one we are looking for.

At the end of the loop, and it is 20 times faster than looking through the entire array, the variable I will have found a pointer to an element very close to the one we are looking for, but not equal. Nothing is achieved if we find the position of the element in the array, because we do not know which one is pointing at it.

Having this element that is smaller, but very close to our number, we can proceed with the usual linked search, which will hopefully be very short.

Unfortunately, unlike the binary search, this method is not infallible, and we might have added N/20 comparisons to the time we were going to spend anyway. Usually this will not be the case (it is actually very rare, but possible).

```
1  REM  ** POINTERS **
10  DIM M(900,1):MIN =  - 9.9E34
15  REM  K STORES THE ELEMENT POINTING AT THE LAS
    T ONE.
20 NIL = 0:M(0,0) = NIL:M(0,1) = 0:FALSE = 0:TRUE
     = 1
30  HOME : VTAB 5: PRINT "1-ADD": PRINT "2-FIND":
     PRINT "3-DELETE": PRINT "4-LIST ";
40  GET W$:V =  VAL (W$): IF V < 1 OR V > 4 THEN
    40
50  ON V GOTO 100,200,300,400
100  PRINT : INPUT "NUMBER TO ADD:";E: GOSUB 5000

110  GOTO 30
200  PRINT : INPUT "NUMBER TO FIND:";E
210  GOSUB 7000: IF J = 0 THEN  PRINT "NOT FOUND
    ": GOTO 230
220  PRINT E" IN POSITION "J
230  GET W$: GOTO 30
300  PRINT : INPUT "NUMBER TO DELETE:";E
310  GOSUB 9000: IF J = 0 THEN  PRINT "NOT FOUND"
    : GOTO 330
320  PRINT "DELETED"
330  GET W$: GOTO 30
400  GOSUB 6000: GET W$: GOTO 30
5000  REM  ** ADD **
5010 N = M(0,1):M(N + 1,1) = E
5020 J = 0
5030  GOSUB 8000
5040  IF M(J,0) = NIL THEN M(J,0) = N + 1:M(N + 1
    ,0) = NIL:M(0,1) = N + 1:K = J: RETURN
5050 L = J:J = M(J,0)
5060 V = M(J,1): IF E > V THEN 5040
5070 M(L,0) = N + 1:M(N + 1,0) = J:M(0,1) = N + 1
    :K = L: RETURN
6000 JJ = 0: PRINT
6010 JJ = M(JJ,0): IF JJ = NIL THEN  PRINT : RETURN

6020  PRINT M(JJ,1): GOTO 6010
7000  REM  ** SEARCH **
7020 J = 0: IF M(J,0) = NIL THEN  RETURN
7030 L = J:J = M(J,0)
7040  IF M(J,1) = E THEN  RETURN
7050  IF M(J,1) > E OR M(J,0) = NIL THEN J = 0: RETURN

7070  GOTO 7030
8000  REM  ** PSEUDO-BINARY SEARCH **
8010  IF M(0,1) = 0 THEN J = 0: RETURN
8020 P = M(0,0):G = MIN:FL = FALSE
8030  FOR I = 1 TO M(0,1) STEP 20
8040 J = M(I,0): IF M(J,1) > E THEN 8070
8050  IF M(J,1) = E THEN  RETURN
```

```
8060  IF M(J,1) > G THEN P = J:G = M(J,1):FL = TR
     UE
8070  NEXT : IF FL THEN J = P: RETURN
8080 J = 0: RETURN
9000  REM  ** DELETE **
9010  GOSUB 7000: IF J = 0 THEN  RETURN
9020 N = M(0,1): IF M(J,0) = N THEN M(L,0) = J: GOTO
     9025
9022 M(L,0) = M(J,0)
9025  IF J = N THEN 9040
9030 M(J,0) = M(N,0):M(J,1) = M(N,1): IF K <  > J
      THEN M(K,0) = J
9040 N = N - 1:M(0,1) = N: FOR I = 0 TO N - 1: IF
     M(I,0) = N THEN K = I: RETURN
9050  NEXT
```

4

FILES

4.1 SEQUENTIAL FILES

When strings are PRINTed to a sequential file, there is no limitation as to what characters are in the string. Quotes, colons, commas, even control characters can be PRINTed normally into a file.

But when these strings are READ back into string variables, the INPUT command can ignore or reject parts of them, making the strings read differently from the ones originally PRINTed.

The main reasons for these differences are: First, INPUT completely ignores all trailing blanks; second, when it encounters either a colon or a comma, it ignores the rest of the string and PRINTs the EXTRA IGNORED message; third, if the string contains a return (CHR$ (13)), the INPUT believes the string is finished and the rest of it is left for the next INPUT to read.

As you can see, the INPUT command behaves exactly the same way when reading from disk as when reading from the keyboard; therefore, the solutions to the above problems apply equally well to both modes.

The easiest way to avoid the EXTRA IGNORED is to enclose your strings in quotes when they are PRINTed. This way, trailing blanks are not ignored, and commas and colons are read into the string exactly as they are written.

A complication arises when you include quotes in the string, because they are considered to be a string terminator, too.

4.2 READING SEPARATORS

The most accurate way to read a string from the disk is using GET instead of INPUT. The process is obviously slower and slightly more complicated, but worth the effort, if you need these characters that are difficult or impossible to handle with INPUT.

Program READ FILE supposes the first thing written in the sequential file TEST (which you will have to create—using the EDITOR it is a matter of seconds) is the number of strings in the file. This is read using the INPUT statement, since numbers don't pose any problem here.

Line 30 starts a loop that will read N strings. Line 40 GETs a character. If it is a <return>, the string is considered to be finished, and the jump to line 70 increments I, so that whatever is read next ends up in the following element of the string array.

If the character read is any other character, it is added to S$ (I), the string currently being READ.

A special case has to be considered here: There is the possibility of reaching 255 characters and still having more to read before a <return> indicates the end of the string. This can happen when several strings are PRINTed to a file with a semicolon at the end, combining them as if they were only one.

Since no more than 255 characters can be in a string, when this limit is reached the string must be terminated and the following characters added to the next element of S$ ( ), even though we are not being faithful to the real contents of the file.

```
1  REM  ** READ FILE **
10 D$ =  CHR$ (4): DIM S$(100)
20  HOME : VTAB 10: HTAB 3: INPUT "FILENAME:";FL$
     : IF FL$ = "" THEN 30
30 W$ =  LEFT$ (FL$,1): IF W$ < "A" OR W$ > "Z" THEN
     20
40  PRINT : PRINT D$"OPEN "FL$: PRINT D$"READ "FL
     $
50  INPUT N: FOR I = 1 TO N
60  GET W$: IF W$ =  CHR$ (13) THEN 90
70  IF  LEN (S$(I)) = 255 THEN 90
80 S$(I) = S$(I) + W$: GOTO 60
90  NEXT
100  PRINT : PRINT D$"CLOSE"
110  FOR I = 1 TO N: PRINT S$(I): NEXT
```

If extreme accuracy is needed, an array of booleans of the same size of S$ ( ) could be DIMensioned, and the corresponding element of it could be set to TRUE, indicating that the string with that index is continued in the next one.

At the end of the loop, all strings have been READ into S$ ( ).

If the strings contain <return>'s which are not to be taken as separators but which must be in the string, the length must be known in advance, or a different character must be used as a separator, so that GET can detect the end of it.

4.3 READING FILES OF UNKNOWN LENGTH

In the previous program, we were lucky the first element in the file was the number of strings to read. This is not always the case; in many situations we don't even know if one string is on the file.

Fortunately, the use of ONERR GOTO solves this problem. When using either the INPUT or the GET statements to read a file, an attempt to read beyond the last field in the file causes an END OF DATA error (error code 5), which can be detected by an error handling routine. The file should then be CLOSEd, and the program can continue normally. If the GET statement is being used to read the strings, the last one being processed should be examined to see if it contains any characters; if it doesn't, the total number of strings should be reduced by one..

4.4 WRITING ON EXISTING FILES

When a sequential file writes over an already existing one, it doesn't erase its previous contents. If the old file was longer than the new one, the result will be a combination of the new one, plus the extra elements of the old one. If the length of the file is not specified, it is a good idea to DELETE the file before starting writing to it.

However, the DELETE command can cause an error if the file to DELETE doesn't exist (and when you are taking this precaution, you want the program, not you, to check if it exists or not); therefore, OPENing the file creates it if it doesn't already exist, and DELETE takes care of it. Short life!

4.5 RANDOM ACCESS FILES

There is no difference between the contents of a sequential and a random access file. They are basically the same, except that the second one can be written or read in a nonsequential way.

If you create a random access file and write to the first and third registers, for example, it is equivalent to having written a sequential file, only some parts of it have not been written yet. And there lies one of the dangers: If in the previous example you tried to read the second register, an END OF DATA error would make your program fail.

To prevent this from happening, the file should be initialized in the moment it is first created. Since a random access file is basically equal to a sequential one, it can be originally OPENed as a sequential file, write to it (what to write is explained later), and CLOSE it.

That way, any attempt to READ a record that has never been written to will not cause the END OF DATA error.

When you PRINT something to a file (whether it be number or string), a <return> is placed after it automatically, unless a semicolon is placed at the end of what is being PRINTed, in which case the next characters sent to the file will be appended to it.

When the file is being READ, every INPUT statement reads all the characters until the first <return>.

If we want to initialize a file so that when it is read there is something to read (to avoid the END OF DATA) and at the same time nothing is read into the variables in the INPUT, the best choice is to fill the file with <return>'s. This solution works with string variables, but not when a number is expected.

Calculate the size of your file by multiplying the record size (the length specified in the L part of the OPEN) by the maximum record you think you will use. OPEN the file for sequential output, and PRINT as many CHR$ (13)'s as the number calculated.

Since register 0 is available, it is a good idea to store there the current number of active records in the file. This number should be READ as soon as the file is OPENed. If an END OF DATA occurs, the file has not been created, and should be initialized (ONERR GOTO is necessary here), but not before asking for confirmation, because the disk in the drive might be the wrong one, and it would be a waste of time creating it there.

4.6 KEEPING FILES OPEN

The slowest instruction when READing or WRITEing a file is OPEN. Therefore, it is a good idea to execute it at the beginning of the program and keep the files open until they are no longer needed. The one problem with this is when a file is OPENed, all the input or output is taken from, or sent to, the file, depending on the way it is set. But with random files, since to access every record a new READ or WRITE instruction must be executed, as soon as the INPUT or PRINT to a particular record has been made, the access to the screen or to the keyboard can be recuperated by PRINTing a ctrl-D (or any nonprinting character), as in program FIXED LENGTH.

This way, not only can one file be OPEN at a particular time, but up to 15 active ones.

4.7 FIXING THE LENGTH OF THE RECORDS

Once a random access file has been OPENed and it has been set to WRITE to a particular record, the behavior is not different from a sequential file. No check is made to see if you actually stick to the length originally specified. Therefore, you can easily exceed that length and then part of your information is recorded at the beginning of the next record. If you read it back, everything appears to be fine, since the READ is performed sequentially too. But if you PRINT something to that next record, you will lose the ending part of the previous one, with confusing results. Besides, you may PRINT a string where the next record expects a number, and an error will occur.

To prevent this from happening, it is a good idea to make sure the length of what is being PRINTed to the file is exactly equal, or less than the one specified when the file was OPENed.

Program FIXED LENGTH RECORD combines some of the features described here.

Line 10 sets D$ for the disk operations, and executes an ONERR GOTO command that will detect an uninitialized file.

Line 20 OPENs the file TEST, and line 25 tries to READ the record #0, where the number of active records is kept. If the file has not been created, an END OF DATA is detected by the ONERR

GOTO. The program branches to line 10000 and then to 10010, where a D$ interrupts the READ from the file, and restores the normal input from the keyboard.

If the disk was the wrong one, lines 10010 and 10020 give you the option of repeating the operation with a new disk. Otherwise, line 10040 sets the WRITE to start at record 0, which places the file pointer at the very beginning of the file. Then, knowing that the length of the record is 42, and calculating that a maximum of 30 records is going to be used, line 10050 PRINTs 30 W$'s, each one of which has been set to contain exactly 42 <return>'s. At this point, the file is treated like a sequential one.

After doing this, a 0 is PRINTed in register 0, indicating that, even though the file has been initialized, 0 records are active at this time. The initialization is then completed.

If line 25 finds a number (even a 0) in the file, line 30 PRINTs the following menu:

```
1-ADD A NAME
2-READ INFORMATION
3-QUIT
```

1-ADD A NAME

Lines 100 through 120 ask for a name, a position description, and a salary. Line 130 sets R (the register number) to N+1, so that the new information is placed right after the last one entered. At the beginning, when N = 0, the N+1 record is number 1, and the information in register 0 is not affected.

Subroutine 1000 uses BF$ as a temporary buffer to store the three strings, each one of which is forced to be of the predetermined length (20 bytes for the name, 10 for the position, and 9 for the salary), exactly as done in output, section 2.1. Notice that BF$ is actually one single string, with no separators in it. Now the only way to recuperate the original strings is to know their original length.

Line 1010 writes BF$ in register R, and line 1030 updates register 0 with the new value of N. Line 1040 releases the output from the disk to the screen, and the subroutine RETURNs.

2-READ INFORMATION

Line 200 asks for the name to look for. Line 210 adds enough spaces to make it of the predetermined length (the way it is in the file) for comparison purposes. Line 220 starts a loop that will scan through the entire file, for as many records as N indicates, or until the name is found.

Subroutine 2000, called by line 220, reads into BF$ the record specified by R, releases the input, and RETURNs.

Line 230 compares N$ with the LEFT$ of BF$. If it finds that they are equal, it doesn't have to read the other pieces of information, because all of it is in BF$ and it can be displayed immediately, as line 240 does. Line 250 calls subroutine 3000 for a pause.

The process is repeated until either the name is found or the end of the file is encountered. If this happens, line 280 informs you that the name is not in the file. Subroutine 3000 is called for a pause after which the program goes back to the menu.

3-QUIT

Before the actual termination of the program, it is important to CLOSE the file. When a WRITE is executed, the information is temporarily stored in a buffer, but not necessarily transferred to the disk. DOS judges the best time to do it, and if you turn off your computer or change the disk before actually writing this information, the next time you try to read it, you will find that it was never on the disk.

Line 300 CLOSEs the file, writing any unwritten information.

```
1  REM  ** FIXED LENGTH RECORD **
5  ONERR  GOTO 10000
10 D$ =  CHR$ (4):BL$ =  CHR$ (7)
20  PRINT D$"OPEN TEST,L42"
25  PRINT D$"READ TEST,R0": INPUT N: PRINT D$
30  HOME : VTAB 9: PRINT "1-ADD A NAME": PRINT : PRINT
     "2-READ INFORMATION": PRINT : PRINT "3-QUIT
     ";
40  GET W$: IF W$ < "1" OR W$ > "3" THEN 40
50  ON  VAL (W$) GOTO 100,200,300
100  HOME : VTAB 10: INPUT "NAME:";N$
110  PRINT : INPUT "POSITION:";P$
120  PRINT : INPUT "SALARY:";S$
130 R = N + 1: GOSUB 1000:N = N + 1
140  GOTO 30
```

```
200  HOME : VTAB 10: INPUT "NAME TO READ:";N$
210 N$ =  LEFT$ (N$ + "                 ",20)
220  FOR R = 1 TO N: GOSUB 2000
230  IF N$ <  >  LEFT$ (BF$,20) THEN 280
240  PRINT : PRINT "POSITION:     "; MID$ (BF$,21,
     10): PRINT : PRINT "SALARY:"; RIGHT$ (BF$,9)

250  GOSUB 3000: GOTO 30
280  NEXT : PRINT : PRINT "NOT FOUND": GOSUB 3000
     : GOTO 30
300  PRINT : PRINT D$"CLOSE": HOME : END
1000 BF$ =  LEFT$ (N$ + "                  ",20
     ) +  LEFT$ (P$ + "         ",10) +  RIGHT$
     ("          " + S$,9)
1010  PRINT D$"WRITE TEST,R"R
1020  PRINT BF$
1030  PRINT D$"WRITE TEST,R0": PRINT R
1040  PRINT D$: RETURN
2000  PRINT D$"READ TEST,R"R
2010  INPUT BF$: PRINT D$: RETURN
3000  PRINT : PRINT : PRINT " PRESS <RETURN> TO C
     ONTINUE ";
3010  GET W$: IF W$ <  >  CHR$ (13) THEN 3010
3020  RETURN
10000  IF  PEEK (222) <  > 5 THEN  POKE 216,0: RESUME

10005  HOME
10010  PRINT D$: PRINT : HTAB 9: PRINT "THE FILE
     'TEST' HAS NOT": PRINT : HTAB 12: PRINT "BEE
     N INITIALIZED.": PRINT : HTAB 6: PRINT "SHOU
     LD IT BE STARTED IN THE": PRINT : HTAB 11: PRINT
     "DISK IN DRIVE ONE ? "BL$;
10020  GET W$: IF W$ <  > "Y" THEN  PRINT : PRINT
     D$"CLOSE": HOME : END
10030 W$ = "": FOR I = 1 TO 42:W$ = W$ +  CHR$ (1
     3): NEXT
10040  PRINT : PRINT D$"WRITE TEST,R0"
10050  FOR I = 1 TO 30: PRINT W$;: NEXT
10060  PRINT : PRINT D$"WRITE TEST,R0": PRINT 0: PRINT
     D$: RUN
```

4.8 PACKING DATA

When space is limited and the amount of data is large, every single byte that can be economized is important. Since it takes one byte to represent every character, not much can be done to save space when using strings. With numbers, there is a way to do some economy.

If we consider only integer numbers, great savings can be had when we pack as many digits as possible into each byte.

As described in section 11.12, Binary Numbers, every byte can take values from 0 to 255. If we have numbers in this range, we can use the CHR$ ( ) function to make the number become a character, and this single character will be able to store even numbers with three digits in only one byte.

If we use two bytes and decide that the first one's value is going to be multiplied by 256 and added to the value of the second one, we will be able to store numbers from 0 to 65535. And if we use four bytes following the same pattern, we can store integer numbers from one to nine digits with perfect accuracy, and if the original number had nine digits, we will be saving five bytes, which multiplied by the number of registers in your file can add up to a considerable saving.

Program PACK DATA initializes TRUE and FALSE to their Applesoft equivalents, and BL$ to the bell character.

Line 10 asks for a number, and line 20 calls subroutine 1000, the place where the encoding (or packing) takes place.

Line 1005 rejects numbers that cannot be stored in the four byte limit. If the number is legal, line 1010 clears X$ (where the encoded number will be stored), and starts the loop that divides N by decreasing powers of 256, just as it is done in section 11.12, with the binary numbers.

In line 1040, S is converted to its CHR$ equivalent, and is added to X$. At the end of the loop, in line 1050, the number has been completely encoded.

Notice that if the first divisions produce a 0, the resulting character is not added to X$, which would make X$ always have four bytes, even with a number of only one digit.

Subroutine 2000 decodes the number, but doesn't use the powers of 256; instead, it multiplies EX (which starts with 1) by 256 every pass of the loop. Both are equivalent, and are included here only to show the two possibilities.

Line 2030 gets the individual characters from X$, and converts them into their original values using the ASC function.

Using these two subroutines, any integer positive number can be packed, and stored in a shorter way.

If a fractional part is needed, the method of fixed decimal explained in program EXTENDED PRECISION OPERATIONS can be used: If you want to store two decimal places, multiply the number by 100 (shifting it two places to the left) and get its INTeger part.

If negative numbers are needed, you can add 5E7 to the number before encoding it, and subtract the same number after decoding it; the accepted range will now be limited to +500000000 to -500000000.

```
1   REM  ** PACK DATA **
10  TRUE = 1:FALSE = 0:BL$ =  CHR$ (7)
20   HOME
30   INPUT "NUMBER:";N
40   GOSUB 1000: PRINT "ENCODED:" LEN (X$)" BYTE";
      : IF  LEN (X$) > 1 THEN  PRINT "S";
50   PRINT
60  N = 0: GOSUB 2000: PRINT "DECODED:";N: PRINT :
       GOTO 30
1000  REM  ** ENCODE **
1005  IF N >  = 1E9 OR N < 0 THEN  PRINT BL$;"ENC
     ODING OVERFLOW": RETURN
1010 X$ = "":FL = FALSE: FOR I = 3 TO 0 STEP  - 1

1020 A = N / 256 ^ I:S =  INT (A):N = N - 256 ^ I
      * S
1025  IF S <  > 0 THEN FL = TRUE
1040  IF FL THEN X$ = X$ +  CHR$ (S)
1050  NEXT : RETURN
2000  REM  ** DECODE **
2010 N = 0: IF  LEN (X$) = 0 THEN  RETURN
2020 N = 0:EX = 1: FOR J =  LEN (X$) TO 1 STEP  -
     1
2030 N = N +  ASC ( MID$ (X$,J,1)) * EX
2040 EX = EX * 256
2050  NEXT : RETURN
```

4.9 CHANGING THE DRIVE, SLOT, AND VOLUME

If any of the three parameters, volume, slot, or drive, are changed by a disk command, the new value remains unchanged until it is reset by another one, even if an error is caused, as when the specified disk drive does not exist, there is nothing on that slot, or the volume number doesn't match the one in the diskette.

These values are stored in three memory addresses which can be PEEKed from within a program, and changed with POKE. These locations are:

43622 Volume
43624 Drive
43626 Slot

4.10 MAXFILES

When DOS is loaded, it automatically reserves space for three disk buffers, which are used to store information temporarily while working with files (EXEC file uses one, too). Each one of these buffers takes 595 bytes.

If you know you are not going to use all of them at a given time, you can instruct DOS to release the memory area originally reserved for those buffers you do not need. This is done with the MAXFILES command.

Since the disk buffers are just above HIMEM, when MAXFILES is executed to reduce their number, it moves HIMEM up, and if any string variables had been used previously in the program, their contents are usually lost (strings are stored starting at HIMEM).

Therefore, it is a good idea to use MAXFILES as the first active instruction of the program.

4.11 READING THE KEYBOARD WHILE WORKING ON THE DISK

As described in program NONSTOP, section 1.13, the keyboard can be read without interrupting the program. When a disk operation takes a long time, like reading or writing to a big file, it is convenient to have the option of stopping the process at any time. If <esc> is chosen to interrupt the disk process and return to the menu, the instruction would be:

```
500 IF PEEK (-16384) = 155 THEN POKE -16368, 0:
GOTO (MENU)
```

Instruction 500 has to be placed inside the loop that does the reading or writing.

If the file is sequential, it must be CLOSEd and reOPENed to start reading or writing from the beginning.

4.12 TURNING OFF DOS

When you need all the memory you can get and you are not going to store anything in the disk, or use DOS in any way at all, it can be disconnected, and the memory normally occupied by it can be used.

When the instructions PR#0 and IN#0 are executed without the D$ used for the disk commands, DOS is disconnected. All the disk commands are no longer recognizable by Applesoft, and an attempt to see the CATALOG, for example, causes a SYNTAX ERROR. But the memory does not increase automatically: It has to be done manually.

HIMEM, that is, the highest memory location available for the user (and therefore by a program), is stored in locations 105 and 106, as usual, least, and most significant bytes, respectively. In systems with 48k bytes in RAM, the highest memory location that can be changed is 49151, but when DOS is "alive," it uses about 10k bytes and reduces this number considerably.

Therefore, if DOS has been disconnected and we POKE into 105 and 106 the equivalent of 49151, all the RAM in the computer is available.

Program PRIME NUMBERS performs a DIM M% (23400), which would cause an OUT OF MEMORY error when used with DOS. That is because in line 5 it did the PR#0 and IN#0, and executed POKE 116,191 and POKE 115,255.

Apart from the incredible increase in memory, program PRIME NUMBERS uses a very fast algorithm.

```
1  REM  ** PRIME NUMBERS **
5  PR# 0: IN# 0: POKE 116,191: POKE 115,255
10  HOME : DIM M%(23400)
20  INPUT "HIGHEST NUMBER:";N
30  FOR I = 2 TO  SQR (N)
40  IF M%(I) = 1 THEN 80
50  PRINT I" ";: FOR J = I * 2 TO N STEP I
60 M%(J) = 1
70  NEXT
80  NEXT I: FOR J = I TO N: IF M%(J) = 0 THEN  PRINT
     J" ";
90  NEXT
```

5

EXEC FILES

When a file is read by the DOS command EXEC, Applesoft accepts every character as if it had been typed at the keyboard.

If the line read starts with a valid Applesoft number (0 to 63999), it is stored in memory as a program line. If there is an instruction in memory with that number already, it is deleted and replaced by the new one.

If it is an immediate mode instruction, Applesoft executes it.

Otherwise, a SYNTAX ERROR message is PRINTed, and EXEC continues until it has read the entire file.

EXEC cannot be stopped by ctrl-C or RESET; the only way to do it is to turn OFF the computer.

This is a very powerful feature, as will be seen in the following section, and in parts of programs discussed later.

5.1 PLOTTING A FUNCTION

If you wanted to write a program to plot a function, you would find one problem: The function has to be entered as a string, and to evaluate the points to plot, the string would have to be either analyzed character by character and the operations or functions simulated (which would be nearly equivalent to writing a new Applesoft interpreter), or the contents of the string would have to be POKEd into memory and used as a regular program line or subroutine, to evaluate the points to plot. This would require knowledge of how and where a program is stored in memory, and how to add an instruction without affecting the rest of the program.

Both procedures are complicated, and fortunately there is another way to do it—using the EXEC command. These are the steps to follow:

1. Enter the function as a string.
2. Create a text file containing the basic body of the program to plot functions.
3. PRINT to the file the function preceded by a line number and followed by a RETURN (a subroutine RETURN). The subroutine thus formed has to be called from the program to get the values of the points to plot.
4. EXEC the file.
5. RUN the program.

These steps are followed in program PLOTTER.

Line 5 initializes D$ for the disk operations, and Q$ to CHR$ (34), a quote. When printing characters into a file, quotes cannot be included directly, because they are taken as the starting and ending points of strings. Therefore, the only way to print them is to use the CHR$ function, which in this case was used to put the quote in Q$.

Line 20 does the INPUT of the body of the function, and line 30 opens the file DUMMY.

```
1  REM  ** PLOTTER **
5 D$ =  CHR$ (4):Q$ =  CHR$ (34)
10  TEXT : HOME : VTAB 10: INPUT "F(X)=";F$
20  PRINT D$;"OPEN DUMMY": PRINT D$;"DELETE DUMMY
   "
25  PRINT D$;"OPEN DUMMY": PRINT D$;"WRITE DUMMY"

30  PRINT "130 VTAB 22:?"Q$;"F(X)=";F$;Q$
40  PRINT "1000 Y=";F$":RETURN"
50  PRINT "DEL 5,70"
60  PRINT "RUN": PRINT D$"CLOSE"
70  PRINT D$"EXEC DUMMY": END
100  HGR : HCOLOR= 3: HOME
110  FOR X = 0 TO 279 STEP 5: HPLOT X,80: NEXT
120  FOR Y = 0 TO 191 STEP 5: HPLOT 140,Y: NEXT
130  VTAB 22: PRINT "F(X)=SIN(X)"
135 X =  - 140: GOSUB 1000: HPLOT X + 140,80 - Y
140  FOR X =  - 139 TO 139: GOSUB 1000
150  HPLOT  TO X + 140,80 - Y: NEXT
160  PRINT "PLOTTING COMPLETED.";: CALL  - 868: END
```

Lines 40 to 50 create the following file (Program Fragment 5.1), assuming the function entered was 50*SIN (X*0.05):

```
1000 Y = 50 * SIN (X * 0.05) : RETURN
15 VTAB 22 : ?"F (X) = 50 * SIN (X * 0.05)"
DEL 20, 60
RUN
```

Program Fragment 5.1.

Line 57 EXECs the file DUMMY: Instructions 1000 and 15 are read and added to the program, and lines 20 to 60 are DELeted (those lines took care of the input and the creation of the file, and are no longer needed).

The program that resides now in memory has some of the original instruction of program PLOT, but instructions 15 and 1000 make it good only for one function, the one you typed in. Program FINAL PLOTTER shows the final version.

```
1  REM  ** PLOTTER **
100  HGR : HCOLOR= 3: HOME
110  FOR X = 0 TO 279 STEP 5: HPLOT X,80: NEXT
120  FOR Y = 0 TO 191 STEP 5: HPLOT 140,Y: NEXT
130  VTAB 22: PRINT "F(X)=50 * SIN(X * 0.05)"
135 X =  - 140: GOSUB 1000: HPLOT X + 140,80 - Y
140  FOR X =  - 139 TO 139: GOSUB 1000
150  HPLOT  TO X + 140,80 - Y: NEXT
160  PRINT "PLOTTING COMPLETED.";: CALL  - 868: END

1000 Y = 50 *  SIN (X * 0.05): RETURN
```

Therefore, a new program has been created, using just some of the elements of the original one. At this point, it can be treated as any Applesoft program.

The next command in DUMMY is RUN, and execution of the program finally begins.

Lines 65 through 150 do the actual plotting of the function. No error checking is made, so bad functions (those producing points outside the range of the HGR) will make the program fail. But the point here is that we have very easily achieved something that would otherwise be much harder to do.

5.2 DATA CREATOR

Many times there is a collection of numbers you would like to have as DATA in your program. If the number is not too large, typing these numbers directly is not a problem, but as the size increases, so does the possibility of error and the time involved.

Now that we know how to create files that when EXECed become part of a program, we can leave this typing to the computer.

Program DATA CREATOR gives you two options: To create data lines directly from memory (you can take a range of memory and convert it into DATA lines in a program), or to read numbers directly from the keyboard and create the data lines.

Line 100 DImensions M ( ) to 100 elements; this is an arbitrary number, and can be changed to any size desired. Line 12 displays a menu of three options:

1-DATA FROM MEMORY
2-DATA FROM KEYBOARD
3-QUIT

FL is set to the value entered.

Option 1 asks in line 15 the starting and ending addresses and sets N (the actual number of items in the DATA list) accordingly.

The second option asks for the number of items to be entered (line 20), and proceeds to do the INPUT of all the numbers.

Line 30 asks for an initial line number (the first line of the DATA in the file produced) and an increment.

Line 45 asks if the number of items should be included in the DATA, and line 48 asks if the starting address should be included too, if the first option was selected.

Line 50 sets A$ to the STR$ of the initial line number followed, by ''DATA'', and adds N if F1 is TRUE (if the answer to line 45 was positive). Line 60 adds the starting address if F2 is true.

Every number will be converted to its equivalent string with the function STR$, so that it can be added to A$, which will store all the numbers of a line of DATA's until it reaches the maximum length that can be read by an EXEC file.

Line 10000 OPENs the file DUMMY. Line 10010 starts a loop that goes from SA to EA (starting and ending addresses in the case of the first option, and 1 to N in the second case). Line 10012 jumps to either line 10013 or 10014, depending on the value of FL, to set X to PEEK (I) or M (I).

X is sent to subroutine 10070 which converts it to a string. If it finds that the length of A$ would exceed 230 characters when added this new string, it PRINTs A$ to the file, taking care to chop off the last comma (with LEFT$ (A$, LEN (A$) – 1)). A$ is emptied, the new line number and the word DATA are put in A$, D is incremented so that it contains the next line number, and the subroutine RETURNs.

If line 10070 finds that A$ can still take at least one more number, W$ (the STR$ of X) is added to A$, and the subroutine RETURNs.

When the I-loop is terminated, in line 10030, A$ might still have a line number with some values that haven't been PRINTed to the file. If so, line 10040 PRINTs A$ (without the last comma).

When the process is finished, typing EXEC DUMMY will bring into memory the lines containing the DATA. The resulting program can then be SAVEd, new lines can be added to it, or it can be merged with another existing program using the program RENUMBER, available in the SYSTEM MASTER diskette.

```
1  REM  ** DATA CREATOR **
10 D$ =  CHR$ (4): DIM M(100):TRUE = 1:FALSE = 0
12  HOME : VTAB 4: PRINT "1-DATA FROM MEMORY": PRINT
     : PRINT "2-DATA FROM KEYBOARD ": PRINT : PRINT
     "3-QUIT ";
13  GET W$: IF W$ < "1" OR W$ > "3" THEN 13
14 FL =  VAL (W$): ON FL GOTO 15,20,100
15  PRINT : PRINT : INPUT "  STARTING ADDRESS : "
     ;SA: PRINT : INPUT "  ENDING ADDRESS : ";EA
16 N = EA - SA + 1: GOTO 30
20  PRINT : PRINT : INPUT "  NUMBER OF ITEMS : ";
     N: FOR I = 1 TO N: PRINT "M("I;: INPUT ")=";
     M(I): NEXT
25 SA = 1:EA = N
30  PRINT : INPUT "INITIAL LINE NUMBER:";IL: IF I
     L < 0 OR IL > 63999 THEN 30
40  PRINT : INPUT "INCREMENT:";IN: IF IN < 1 THEN
     40
45 F1 = FALSE: PRINT : PRINT "SHOULD THE NUMBER O
     F ITEMS BE": INPUT "THE FIRST ELEMENT OF THE
      LIST ? ";W$: IF  LEFT$ ("YES", LEN (W$)) =
     W$ THEN F1 = TRUE
47  IF FL = 2 THEN 50
```

```
48  PRINT : PRINT "SHOULD THE STARTING ADDRESS BE
      THE": INPUT "SECOND ELEMENT OF THE LIST ? "
     ;W$: IF  LEFT$ ("YES", LEN (W$)) = W$ THEN F
     2 = TRUE
50 D = IL + IN:A$ =  STR$ (IL) + " DATA": IF F1 THEN
     A$ = A$ +  +  STR$ (N) + ","
60  IF F2 THEN A$ = A$ +  STR$ (SA) + ","
70  GOTO 10000
100  HOME : END
10000  PRINT : PRINT D$"OPEN DUMMY": PRINT D$"DEL
     ETE DUMMY": PRINT D$"OPEN DUMMY": PRINT D$"W
     RITE DUMMY"
10010  FOR I = SA TO EA
10012  ON FL GOTO 10013,10014
10013 X =  PEEK (I): GOTO 10020
10014 X = M(I)
10020  GOSUB 10070
10030  NEXT
10040  IF A$ <  > "" THEN  PRINT  LEFT$ (A$, LEN
     (A$) - 1)
10050  PRINT D$"CLOSE"
10060  END
10070 W$ =  STR$ (X): IF  LEN (W$) +  LEN (A$) >
     230 THEN  PRINT  LEFT$ (A$, LEN (A$) - 1):A$
      =  STR$ (D) + "DATA" + W$ + ",":D = D + IN:
      GOTO 10090
10075  IF D > 63999 THEN  PRINT D$"CLOSE": PRINT
     : PRINT "LINE NUMBER EXCEDED 63999 ": END
10080 A$ = A$ + W$ + ","
10090  RETURN
```

5.3 PROGRAM MODIFICATION

This example incorporates some interesting features of EXEC files. Its purpose is to read a program, convert it to its ASCII equivalent, and then allow you to study and change its parts.

To load the ASCII version of the program you want to study, you could LOAD it into memory, add some lines to it so that it OPENs a file to WRITE, and LISTs itself.

Program PROGRAM MODIFIER does those steps for you; since LOADing a program into memory erases the previous one, this process has to be done using EXEC files.

Line 20 asks for a program name; line 22 OPENs the file DUMMY, DELETEs it and OPENs it once more. Supposing you provided WORKFILE as your program name, lines 23 through 26 write to the disk the following lines:

```
LOAD WORKFILE
0 D$ = CHR$ (4) : ? : ?D$"OPEN DUMMY" :
  ?D$"DELETE DUMMY" : ?D$"OPEN DUMMY" :
  ?D$"WRITE DUMMY" : POKE 33, 33 : LIST :
  ?D$"CLOSE" : ?D$"EXEC DUMMY" : END
RUN
```

Program Fragment 5.2.

```
LOAD PROGRAM MODIFIER
DEL 0, 30
RUN
```

Program Fragment 5.3.

As soon as the two files have been written, line 30 EXECs the file DUMMY (program fragment 5.2); the first instruction found LOADs program WORKFILE, erasing PROGRAM MODIFIER from memory.

The second line read is line number 0 which is added to the program WORKFILE. If there was a line number 0 already, it is deleted and replaced by this new one.

The next line tells Applesoft to RUN the program in memory. Even though this is the program you wanted to examine and change, now it starts with new instructions that create an image of itself in the disk, with the name DUMMY (so the old DUMMY is replaced by the ASCII version of program WORKFILE). The actual program WORKFILE is not actually executed, because the end of line number EXECs another file, DUMMY1 (program fragment 5.3). PROGRAM MODIFIER is LOADed once more, and its lines 0 through 30 are DELeted from memory. Now (finally) it RUNs.

The variable I, which will be used as the index for the array where the text form of the program will be stored, is set to −1, so that the first instruction is read into the 0th element. (There is an initial <return> that will make the first line read a null string.) That instruction is line 0, which was not originally in the program; this way it is discarded.

PR$ ( ) will store the text version of the program, and will never be changed. That way, after any number of alterations, the original can still be recuperated.

NP$ ( ) will store a copy of PR$ ( ), and it is there that the actual changes are going to be made.

Since the length of the program is unknown an ONERR subroutine is turned ON. That way, when the end of the file is found, execution continues normally.

Line 45 OPENs DUMMY (which after all its changes, now contains the ASCII version of program WORKFILE, plus line 0). Line 50 GETs character by character, and increases the index (I) if a <return> is detected. Unlike previous programs in which a similar read cycle had to watch for strings longer than 255 characters, DUMMY can never have that problem, since a program line cannot have more than 255 characters.

An END OF DATA error makes the program jump to line 370, which CLOSEs the file and branches to line 80. If the last line read is not empty, N gets the value in I (the number of lines read); otherwise, N gets I-1.

Line 107 copies PR$ ( ) into NP$ ( ), and line 110 shows the following menu:

1-LIST
2-FIND/REPLACE
3-CANCEL CHANGES
4-SAVE
5-QUIT

1-LIST

Line 400 PRINTs NP$ ( ), not PR$ ( ), in the screen; therefore, what is always shown is the modified version. Subroutine 550 waits until <return> is pressed.

2-FIND/REPLACE

Line 150 asks for the string you want to find in the program, and stores it in S2$. If you want to look for a variable name to see if it has already been used, you would answer with the variable's name. Then, the program asks for the "STRING TO REPLACE"; if you answer with "&&", no replacement is done. In both cases, LN is set to the length of the string that will be looked for, or will be replaced.

In line 152, if option 1, entire file, is selected, FL is set to TRUE, and the message "PROCESSING" is PRINTed, while the replacement takes place. If option 2 is selected, line 155 PRINTs two lines of instructions for the manual FIND/REPLACE procedure, and execute a POKE 34, 2, so that the two lines are not erased from the screen when it is scrolled.

Line 160 starts the loop that will go through the entire array NP$ ( ), and sets ST to 1. Line 170 copies the current line (NP$ (I)) into S1$, and the program branches to subroutine 320.

Lines 320 through 360 are the subroutine that checks if S2$ is contained in S1$, starting at position ST. This subroutine is so useful and commonly used, that it is included as a standard function in many versions of BASIC.

The behavior is as follows:

If S2$ is a substring of S1$ (or S1$ itself), the variable IN returns with the position of S1$ where S2$ was found.

If it is not found, IN returns with a value 0.

Since a string can be contained more than once in another one, the parameter ST allows search starting at a position other than the beginning of S1$. If ST has a value equal to or smaller than 0, it is assigned the default first position in the string S1$.

On return from subroutine 370, line 170 sees if IN = 0, in which case S2$ was not found in S1$, and there is no use in studying that line any further: A jump to line 270 examines the next line, or terminates the FOR loop.

Assuming that the string found will be replaced by S2$, ST is set to the position after the last character in S2$ in line 175. If FL is TRUE (entire file), the program jumps to line 200, because lines 180 through 194 are used to show the place where S2$ was found in S1$.

Line 180 PRINTs the characters to the left of where S2$ was found; if it is found in position 1, nothing is PRINTed by this line.

Line 190 PRINTs S2$ in INVERSE, and the remaining characters, if any.

Line 192 GETs W$; if R$ indicates that no replacements are to be made, the program jumps to line 255, where W$ is analyzed. Otherwise, lines 200 and 210 replace S2$ for R$ in NP$ (I), in the position where S2$ was found. If the entire file option was selected, line 215 jumps to 260. Otherwise, line 220 PRINTs NP$ (I), and jumps to line 260.

When <esc> is pressed, line 255 sets the window back to normal, calls subroutine 550 for a pause until <return> is pressed, and jumps to the menu.

Otherwise, line 260 examines ST. If it is not at the end of NP$ (I), S2$ might still be contained in the rest of S1$, and the process is repeated starting at 170, with the new search position in ST.

If S1$ has been completely examined, the NEXT in line 270 starts the cycle with the next program line, or terminates the loop, in which case the window is set back to normal, subroutine 550 called for a pause, and returns to the menu.

3-CANCEL CHANGES

Line 100 asks for confirmation. If it is not given, line 110 returns to the menu. Otherwise, line 107 copies PR$ ( ) into NP$ ( ), erasing all the changes made.

4-SAVE

Lines 450 through 470 PRINT NP$ ( ) (Not PR$ ( )) to the file DUMMY, which when EXECed will leave in memory the new program.

```
1  REM  ** PROGRAM MODIFIER **
10 D$ =  CHR$ (4):Q$ =  CHR$ (34): DIM PR$(400)
20  HOME : VTAB 6: INPUT "PROGRAM :";FL$: IF FL$ =
     "" THEN  HOME : END
22  PRINT : PRINT D$"OPEN DUMMY": PRINT D$"DELETE
      DUMMY": PRINT D$"OPEN DUMMY": PRINT D$"WRIT
     E DUMMY"
23  PRINT "LOAD "FL$: PRINT "O D$=CHR$(4):?:?D$"Q
     $"OPEN DUMMY"Q$":?D$"Q$"DELETE DUMMY"Q$":?D$
     "Q$"OPEN DUMMY"Q$":?D$"Q$"WRITE DUMMY"Q$;
24  PRINT ":POKE33,33:LIST:?D$"Q$"CLOSE"Q$":?D$"Q
     $"EXEC DUMMY1"Q$":END": PRINT "RUN"
25  PRINT D$"CLOSE": PRINT D$"OPEN DUMMY1": PRINT
     D$"DELETE DUMMY1": PRINT D$"OPEN DUMMY1": PRINT
     D$"WRITE DUMMY1"
26  PRINT "LOAD PROGRAM MODIFIER": PRINT "DEL 0,3
     0": PRINT "RUN"
30  PRINT D$"CLOSE": PRINT D$"EXEC DUMMY": END
40 I =  - 1:D$ =  CHR$ (4):TRUE = 1:FALSE = 0: DIM
     PR$(200),NP$(200): ONERR  GOTO 370
45  PRINT : PRINT D$"OPEN DUMMY": PRINT D$"READ D
     UMMY"
```

```
50  GET W$: IF W$ =  CHR$ (13) THEN 70
60 PR$(I) = PR$(I) + W$: GOTO 50
70 I = I + 1: GOTO 50
80  IF PR$(I) <  > "" THEN N = I: GOTO 107
90 N = I - 1: GOTO 107
100  PRINT : PRINT : PRINT "CANCEL CHANGES ? " CHR$
     (7);
103  GET W$
104  IF W$ <  > "Y" THEN 110
107  FOR I = 1 TO N:NP$(I) = PR$(I): NEXT : HOME
     : VTAB 6
110  POKE 33,40: HOME : VTAB 6: PRINT "1-LIST": PRINT
     : PRINT "2-FIND/REPLACE": PRINT : PRINT "3-C
     ANCEL CHANGES": PRINT : PRINT "4-SAVE": PRINT
     : PRINT "5-QUIT ";
120  GET W$: IF W$ < "1" OR W$ > "5" THEN 120
130  ON  VAL (W$) GOTO 400,150,100,450,500
150  HOME : VTAB 6: INPUT "STRING TO FIND ";S2$: PRINT
     : INPUT "REPLACE BY:";R$: IF R$ = "&&" THEN
     LN =  LEN (S2$): GOTO 155
152 LN =  LEN (R$): PRINT "1-ENTIRE FILE": PRINT
     : PRINT "2-MANUAL ";
153  GET W$: IF W$ = "1" THEN FL = TRUE: HOME : VTAB
     10: HTAB 16: PRINT "PROCESSING": GOTO 160
154 FL = FALSE
155  HOME : HTAB 8: PRINT "<ESC>:MENU - SPACE:REP
     LACE": HTAB 13: PRINT "OTHER:FIND AGAIN": POKE
     34,2
160  FOR I = 1 TO N:ST = 1
170 S1$ = NP$(I): GOSUB 320: IF IN = 0 THEN 270
175 ST = IN +  LEN (S2$): IF FL THEN 200
180  IF IN > 1 THEN  PRINT  LEFT$ (NP$(I),IN - 1)
     ;
190  INVERSE : PRINT S2$;: NORMAL : IF ST <  =  LEN
     (S1$) THEN  PRINT  RIGHT$ (NP$(I), LEN (NP$(
     I)) - ST + 1);
192  PRINT : PRINT : GET W$: IF R$ = "&&" THEN 25
     5
194  IF W$ <  > " " THEN 255
200 NP$(I) = "": IF IN > 1 THEN NP$(I) =  LEFT$ (
     S1$,IN - 1)
210 NP$(I) = NP$(I) + R$: IF ST <  =  LEN (S1$) THEN
     NP$(I) = NP$(I) +  RIGHT$ (S1$, LEN (S1$) -
     ST + 1)
212 ST = ST +  LEN (R$) -  LEN (S2$)
215  IF FL THEN 260
220  PRINT NP$(I): PRINT : GOTO 260
250  PRINT : GET W$
255  IF W$ =  CHR$ (27) THEN  POKE 34,0: GOTO 110

260  IF ST <  =  LEN (NP$(I)) + 1 THEN 170
270  NEXT I: POKE 34,0: GOSUB 550: GOTO 110
```

```
310  HOME : END
320  IF ST < 0 THEN  RETURN
330  IF ST = 0 THEN ST = 1
340  FOR IN = ST TO  LEN (S1$) -  LEN (S2$) + 1
350  IF  MID$ (S1$,IN, LEN (S2$)) = S2$ THEN  RETURN

360  NEXT :IN = 0: RETURN
370  IF  PEEK (222) = 5 THEN  PRINT D$"CLOSE": GOTO
     80
380  POKE 216,0: RESUME
400  HOME : FOR I = 1 TO N: PRINT NP$(I): NEXT : GOSUB
     550: GOTO 110
450  PRINT : PRINT D$"OPEN DUMMY": PRINT D$"DELET
     E DUMMY": PRINT D$"OPEN DUMMY": PRINT D$"WRI
     TE DUMMY"
460  FOR I = 1 TO N: PRINT NP$(I): NEXT
470  PRINT D$"CLOSE": GOTO 110
500  HOME : END
550  PRINT : PRINT "     PRESS <RETURN> ";
560  GET W$: IF W$ <  > CHR$ (13) THEN 560
570  RETURN
```

5.4 ADDITIONAL USES OF EXEC FILES

EXEC files are used in the following programs:

7.10	Text editor
8.3	Low resolution editor
10.6	Recorder
11.2	Recovering from NEW
11.5	Hidden catalog
11.6	Hidden character creator

6

SCREEN

6.1 BASIC SWITCHES

Eight memory locations control the display of text, and low and high resolution graphics. They are called switches because they can change the display mode just like a switch does, and are called "soft" because they are controlled from software. Table 6.1 shows their address in memory (both numbers are equivalent), and their action.

Table 6.1. Soft switches for the display modes.

LOCATION		ACTION
49232	-16304	GRAPHICS MODE
49233	-16303	TEXT MODE
49234	-16302	FULL PAGE
49235	-16301	MIXED TEXT AND GRAPHICS
49236	-16300	PAGE ONE
49237	-16299	PAGE TWO
49238	-16298	LOW RESOLUTION
49239	-16297	HIGH RESOLUTION

Table 6.2 shows the memory range for both pages of text and low resolution, and high resolution.

Table 6.2. Screen addresses.

MODE	ADDRESS			
	DECIMAL		HEXADECIMAL	
	starts	ends	starts	ends
Text/low res. page 1	1024	2047	$400	$7FF
Text/low res. page 2	2048	3071	$800	$BFF
High res. page 1	8192	16383	$2000	$3FFF
High res. page 2	16384	24575	$4000	$5FFF

6.2 VIDEO SCREEN

The information displayed on the screen in both TEXT and LOW RESOLUTION GRAPHICS shares the same area in memory.

To see how it is actually distributed, RUN the following instruction:

```
10 FOR I = 1024 TO 2047 : POKE I, 193 : NEXT
```

As you can see, it is not linear. In addition, not every byte in the video memory is displayed on the screen. Every 120 bytes a gap of eight bytes is not shown.

The addresses of the beginning byte of every line are given in program COPY SCREEN.

6.3 COPYING THE SCREEN

The easiest and fastest way to copy the entire text/low resolution screen is using the disk command BSAVE (and load it back with BLOAD), using as parameters the beginning address of the video memory (1024), and its length. The length that corresponds to the last byte displayed is 1024+1015, since the last eight bytes are not displayed, but 1024 is easier to remember, and its use does not have any side effect.

If you chose SCREEN as the name of the file where you will store the screen, the following examples make a copy of the text/low resolution screen when the command is issued:

```
BSAVE SCREEN, A1024, L1024            (In direct mode), and
? D$"BSAVE SCREEN, A1024, L1024"      (from within a program)
```

The A and L parameters of BSAVE and BLOAD can be given in hexadecimal, and they are much easier to remember. The above instructions become:

```
BSAVE SCREEN, A$400, L400
? D$"BSAVE SCREEN, A$400, L400".
```

The command BLOAD SCREEN will read and display the screen as it was when it was BSAVEd.

6.4 DISPLAYING LOW RESOLUTION

Since the low resolution screen is taken from the same memory area as the text screen, when the command to BSAVE is used to store low resolution graphics, the image in the disk has no difference from a file saved while in the text mode.

When this file is BLOADed, the screen will be displayed in the current mode, and if it is not the same as when the image was BSAVEd, the image will be seen as random characters in NORMAL, INVERSE, or FLASH. If the low resolution screen was in the mixed graphics/text mode when the file was created and the mode when it is BLOADed does not correspond, the text lines will be displayed as randomly colored and positioned blocks. The same will happen if a text screen was BSAVEd and it is BLOADed while in low resolution graphics mode.

6.5 HIGH RESOLUTION SCREEN

When HGR is executed, page one of high resolution is cleared, the HCOLOR is set to black (0) and the mixed TEXT and HIGH RESOLUTION mode is assumed. Even though the 280 * 192 coordinates are still accepted as valid, an area only 280 * 160 is displayed. When the full page is wanted, POKE -16302, 0 turns OFF the TEXT part, and displays the entire page one. To switch back to the mixed mode, a POKE -16301, 0 must be used.

There is a small difference between the action of GR and HGR: Since the text/low resolution memory area is the same, GR executes automatically a POKE 34, 20, which sets the top margin of the TEXT page to line 20, to avoid interference between graphics and text. HGR doesn't touch location 34, which is a bit disturbing, since a PRINT might send the characters to an area outside the four lines of text, where they are not visible. Therefore, it is a good idea to always follow HGR by a POKE 34, 20 : HOME when the four lines of text are going to be used.

To return to TEXT mode, the commands TEXT and POKE −16303, 0 are nearly equivalent, with the small difference that TEXT sends the cursor to line 24 aside from returning to text. Unlike HGR and HGR 2, TEXT doesn't clear the screen. The complement of the graphic-to-text switch is −16304, which performs a very important task: It switches to high resolution graphics without clearing the screen.

Finally, locations −16300 and −16299 display pages one and two of either mode.

There is a difference between HGR and HGR2 worth remembering: The second one clears the screen and sets high resolution graphics, page two, with no lines of text. This makes sense, since page two of graphics is normally lodging the Applesoft program, and displaying it would show a meaningless collection of rubbish. HGR defaults to four lines of text.

6.6 ACTIVITY SWITCH

When high resolution is entered by means of HGR or HGR2, any action taken in high resolution (apart from HPLOTting, the screen can be cleared without reissuing HGR; there are also the DRAW and XDRAW commands, which are explained later) will affect the corresponding page. When the page switch is used, Applesoft continues drawing or erasing in the previous page (and it has a purpose). Location 230 stores a value related with the page where this action will take place: when 32 is stored in it, everything goes to page one; when 64, page two is affected.

From now on, location 230 will be called the ACTIVITY SWITCH, since it actually controls the destination of every action in the high resolution pages.

6.7 CLEARING HIGH RESOLUTION

Two routines are available to clear the current high resolution page: A CALL 62450 clears the screen to black, and a CALL 62454 clears it to the most recently HPLOTted color. These routines can be used to blank one page while the other is displayed (by directing the clear-screen using location 230), or to fill a page with a color much faster than using HPLOT's.

6.8 FINDING THE HCOLOR

Location 228 contains the current HCOLOR. The value for every color is given in Table 6.3. The values from zero to 255 not included in the table produce combinations of colors.

Table 6.3. HCOLOR internal codes.

VALUE	HCOLOR CODE
0	0
42	1
85	2
127	3
128	4
170	5
213	6
255	7

6.9 SWITCHING BETWEEN MODES

When one of the screen modes changes to another, there is usually a small lapse in which the screen displays the new screen, and cleans it.

If you are in low resolution graphics mode, for example, and the command TEXT is executed, the screen is seen full of INVERSE, NORMAL, and FLASHing characters that are very quickly cleared by a HOME (hopefully), but that produce a very poor impression for the short lapse they stay on the screen.

The same happens when modes are switched, unless the screen that is shown as a result of the operation was already clean before the command was executed.

The best way to solve this problem and have your programs show a smooth behavior is to clean the screen before you issue the command. Here are the various possibilities:

1-TEXT TO LOW RESOLUTION

Since these two modes share the same area in memory, low resolution cannot be cleared before the switch, because any change in the low resolution screen will be visible while in text. Therefore, the best solution is to clear one of the high resolution pages (using the CALL 62450), having set the activity switch so that the clearing actually takes place in the first page. Then, a switch to page one of high resolution is performed, and while the low resolution screen is not visible, it can be cleared using a CALL −1998. Now the change to low resolution finds a clear screen.

2-LOW RESOLUTION TO TEXT

The procedure is the same, in reverse.

3-TEXT TO HIGH RESOLUTION

The activity switch is set so that the desired page is cleared. Then, a CALL 62450 clears the chosen page, and the change to high resolution finds a screen of the desired color.

4-HIGH RESOLUTION TO TEXT

A HOME while in high resolution clears the text screen, providing the window is set to full page. If you suspect (or if you don't mind four extra instructions), reset the window to its normal condition, do a HOME, and then a TEXT. The change is not noticed, but unfortunately TEXT sets the cursor to the last line of the screen (equivalent to VTAB 24). To put the cursor back in the home position (unless you are going to put it somewhere else), repeat the HOME or do a HTAB 1 : VTAB 1.

7

TEXT

7.1 PRINTING THE TEXT SCREEN

Program SCREEN DUMP takes whatever is being displayed in the screen, and sends it to the printer.

Line 3 DIMensions SC$ ( ) and SP ( ). SC$ ( ) will store a copy of the screen, in 24 lines of 40 characters. SP ( ) will contain the address, in memory, of the first byte of every row. SP (1) will be the address of the first byte of line 1, SP (2) of line 2, etc.

Line 4 READs these addresses and line 5 has the DATA. The lines of the actual screen dump have been numbered 1000 and above in increments of 1, so that they can be added to any other program. That will be discussed later.

Line 1001 starts two loops: the I-loop (1 through 24) is going to point at the rows of the screen, and the J-loop (0 through 39) at the columns. Line 1002 assigns to W$ the CHR$ of the character at location I,J, which is found by PEEKing into it using SP (I) for the start of the line and adding J to this address.

If the character is inside the displayable range, it is added to SC$ ( ); otherwise, SC$ ( ) gets a blank. This prevents control characters that might be in the screen, and would make the printer behave in an unpredictable way. (If you want to include INVERSE and FLASH characters in the image of the printer, read section 7.2.)

When every location has been examined, the printer is turned ON with PR#1 in line 1005. All SC$ ( ) is PRINTed, and control is returned to the screen with PR#0.

The usual characters (in NORMAL mode) are stored in the video memory adding 128 to their ASCII code. In other words, an "A" would be found as the number 193 (ASC ("A") = 65).

Sometimes you might want to keep a copy of how the screen looked at a particular moment. This is useful when you are writing the documentation of a program, and you want to show exactly how the screen looks at certain steps of the program.

If you are using program SCREEN DUMP, or even if you have it as a subroutine in your program, normally you would have to stop the normal execution either with Reset or ctrl-c and therefore end up with a BREAK IN <linenumber> on the screen, or you would have to add a GOSUB every time you wanted to dump the screen to the printer. An easy way to solve that is if an ONERR GOTO is included at the beginning of the program, every time ctrl-c is pressed, it can be detected by the error routine and control can be transferred temporarily to the screen dump.

```
1  REM  ** SCREEN DUMP **
2 D$ =  CHR$ (4)
3  DIM SC$(24),SP(24)
4  FOR I = 1 TO 24: READ SP(I): NEXT
5  DATA  1024,1152,1280,1408,1536,1664,1792,1920,
    1064,1192,1320,1448,1576,1704,1832,1960,1104
    ,1232,1360,1488,1616,1744,1872,2000
1000  REM  ** SCREEN DUMP **
1001  FOR I = 1 TO 24: FOR J = 0 TO 39
1002 W$ =  CHR$ ( PEEK (SP(I) + J) - 128): IF W$ >
      = " " AND W$ <  = "^" THEN SC$(I) = SC$(I) +
     W$: GOTO 1004
1003 SC$(I) = SC$(I) + " "
1004  NEXT :X =  PEEK ( - 16336): NEXT
1005  PRINT : PRINT D$"PR#1": PRINT  CHR$ (9)"80N
     "

1006  FOR I = 1 TO 24
1007  PRINT SC$(I): NEXT
1008  PRINT D$"PR#0"
1009  RETURN
```

7.2 CHANGING THE SCREEN ATTRIBUTE

It is not possible to change the attribute (whether the characters are in NORMAL, INVERSE, or FLASH mode) of a portion of the screen from Applesoft directly.

If one title is flashing and you want to turn it into INVERSE, you must PRINT the title again in INVERSE mode exactly over the old one; but if you don't know in advance what is going to be there, changing it is not so easy. By now, it is no problem PEEKing or POKEing in a specific address of the screen. All we need to know is how to change the characters in that position to a different mode.

There are four modes in which the characters can be in video memory: All the modes begin with "@" and end with "?". Values from 0 to 63 are displayed in INVERSE. FLASH has values from 64 to 127.

Two NORMAL modes are available: One has values from 128 to 191, and another from 192 to 255. When a character is sent to the screen with the PRINT command, it is POKEd with the second mode, namely from 192 to 255.

To change the attribute of a part of memory, we must be able to determine which mode it is in before changing it, or else there is the danger of trying to POKE illegal values into memory.

Program SCREEN ATTRIBUTE READs the locations of the leftmost bytes of the rows of video memory into M% ( ), and five values into L( ). Since we are going to change specific titles, those lengths have been stored there.

Line 50 displays a fake menu with five different options, only for the demonstration.

Line 70 GETs a value from 1 to 5, and line 80 calculates the line number of the choice. Subroutine 10000 sets the HTAB and VTAB for the titles and since the uppermost was printed in line 8, when 1 is pressed, the formula multiplies 1 by 2 and adds 6, to get 8, the line number desired. The choices are multiplied by 2 because the titles are separated by one blank line.

L gets the length of the title chosen, and line 81 calls subroutines 1000, 1100, or 1200 depending on the value of FL. Subroutine 1000 changes the title to INVERSE, 1100 to FLASH, and 1200 to NORMAL. To show how these changes can be made starting with any condition, FL will go from 1 to 2 and 3, and the cycle will be repeated over and over.

Line 1000 starts an I-loop which will go from X-1 to X+L–2. The difference between X and I is that HTAB starts in 1 (HTAB 0 is not illegal, but not the first column: it is actually equivalent to HTAB 256, or 6 lines and 15 spaces) and the PEEK has to be made considering the leftmost column as # 0.

Since the number must be in INVERSE mode (from 0 to 63), line 1020 subtracts 64 as many times as necessary until the number found in the position is in the proper range. Line 1030 POKEs it back where it was taken. Since the loop covers all the characters of the title, it all becomes characters in INVERSE mode.

Line 1100 starts the I-loop for the FLASH mode subroutine. The proper location in memory is PEEKed into in line 1110. Since characters in FLASH mode go from 64 to 127, the numbers are adjusted to this range in lines 1120 and 1130. Line 1140 POKEs the converted value back into memory.

Subroutine 1200 converts the numbers to values above 127, which are in NORMAL mode. If those numbers were to be read from memory into a string, it would be preferable to convert them to 192 and higher, which is NORMAL mode too, but which is exactly how they are POKEd into memory by the PRINT command.

If the length of the title is not known beforehand, it is easy to look for the first nonblank character from the right of the line. That will be the last location to change, unless there are more words or titles in the same line, in which case additional information is necessary.

```
1  REM  ** ATTRIBUTE **
10  DIM M%(24),L(5): FOR I = 1 TO 24: READ M%(I):
      NEXT
20  FOR I = 1 TO 5: READ L(I): NEXT :FL = 1
30  DATA  1024,1152,1280,1408,1536,1664,1792,1920
     ,1064,1192,1320,1448,1576,1704,1832,1960,110
     4,1232,1360,1488,1616,1744,1872,2000
40  DATA  9,12,8,10,4
50  HOME :X = 4:Y = 8: GOSUB 10000: PRINT "1-BEGI
     NNING":Y = Y + 2: GOSUB 10000: PRINT "2-CONT
     INUATION"
60 Y = Y + 2: GOSUB 10000: PRINT "3-APPROACH":Y =
     Y + 2: GOSUB 10000: PRINT "4-FINAL MILE":Y =
     Y + 2: GOSUB 10000: PRINT "5-END! ";
65 X = 6
70  GET W$: IF W$ < "1" OR W$ > "5" THEN 70
80 V =  VAL (W$):Y = V * 2 + 6:L = L(V)
81  ON FL GOSUB 1000,1100,1200:FL = FL + 1: IF FL
      = 4 THEN FL = 1
90  VTAB 21: HTAB 3: PRINT "HERE THE PROGRAM CONT
     INUES"
100  FOR I = 1 TO 2000: NEXT : GOTO 70
1000  FOR I = X - 1 TO X + L - 2
```

```
1010 W =  PEEK (M%(Y) + I)
1020  IF W > 63 THEN W = W - 64: GOTO 1020
1030  POKE M%(Y) + I,W: NEXT : RETURN
1100  FOR I = X - 1 TO X + L - 2
1110 W =  PEEK (M%(Y) + I)
1120  IF W < 64 THEN W = W + 64: GOTO 1140
1130  IF W > 127 THEN W = W - 64: GOTO 1130
1140  POKE M%(Y) + I,W: NEXT : RETURN
1200  FOR I = X - 1 TO X + L - 2
1210 W =  PEEK (M%(Y) + I)
1220  IF W < 127 THEN W = W + 64: GOTO 1220
1230  POKE M%(Y) + I,W: NEXT : RETURN
10000  HTAB X: VTAB Y: RETURN
```

7.3 SCROLLING THE SCREEN

When the cursor is at the bottom of the screen and a PRINT takes place, the screen is scrolled up. In other words, every line is moved up to the next one, and the last one becomes an empty line. Now that we have the exact positions in memory of every row and column of the screen, it is easy to do a scroll down, left or right.

Program SCROLL DIMensions in line 10 the usual M% (24) to store the position of every row, and it READs them.

Line 15 PRINTs a menu that shows the numbers corresponding to the different scrolls: up, down, left, or right. The scroll up is included here because at times it is needed and, instead of sending the cursor to line 24 and doing a PRINT, a machine language routine is used, since it is already available in the system.

Line 30 GETs a number from 1 to 5, and transfers control to the different routines:

1-SCROLL UP

Line 50 does a call to –912, the scroll-up routine. Since it is a machine language routine, it is very fast; the other three, written in Applesoft, will be much slower.

2-SCROLL DOWN

Line 100 POKEs, line by line, the locations of the previous line, using M% (I+1) and M% (I). In line 110, the top line is cleared, or else the final result would be to have the first line duplicated.

3-SCROLL RIGHT

Line 200 uses the J-loop to move every byte one place to the right. Line 220 clears the leftmost column (160, the value POKEd, is the space in NORMAL mode).

4-SCROLL LEFT

Lines 300 and 310 copy every byte to the left, and line 320 puts blanks in the rightmost column.

The scroll of the entire screen is a slow process, since 960 locations must be changed. Sometimes, though, a scroll in a small window is needed, like in the games in which part of the screen is used to display some table or tables of the action taking place, and a small screen is left for input of commands and output of messages. When a scroll up is needed, the window set with the four locations available for that purpose (locations 32 through 35) is ideal. However the routines in program SCROLL can move the information in a small area in any direction, giving the impression of a window being moved horizontally or vertically.

```
1   REM   ** SCROLL **
10   DIM M%(24): FOR I = 1 TO 24: READ M%(I): NEXT

15   HOME : VTAB 8: PRINT "1-SCROLL UP": VTAB 10: PRINT
      "2-SCROLL DOWN": VTAB 12: PRINT "3-SCROLL RI
      GHT": VTAB 14: PRINT "4-SCROLL LEFT": VTAB 1
      6: PRINT "5-QUIT ";
30   GET W$:V =  VAL (W$): IF V < 1 OR V > 5 THEN
      30
40   ON V GOTO 50,100,200,300,400
50   CALL  - 912: GOTO 30
100   FOR I = 23 TO 1 STEP  - 1: FOR J = 0 TO 39: POKE
      M%(I + 1) + J, PEEK (M%(I) + J): NEXT : NEXT

110   FOR I = 0 TO 39: POKE 1024 + I,160: NEXT
120   GOTO 30
200   FOR J = 39 TO 1 STEP  - 1: FOR I = 1 TO 24
210   POKE M%(I) + J, PEEK (M%(I) + J - 1)
220   NEXT : NEXT : FOR I = 1 TO 24: POKE M%(I),16
      0: NEXT
230   GOTO 30
300   FOR J = 0 TO 38: FOR I = 1 TO 24
310   POKE M%(I) + J, PEEK (M%(I) + J + 1): NEXT :
       NEXT
```

```
320  FOR I = 1 TO 24: POKE M%(I) + 39,160: NEXT
330  GOTO 30
400  HOME : END
1000  RETURN
10000  DATA  1024,1152,1280,1408,1536,1664,1792,1
     920,1064,1192,1320,1448,1576,1704,1832,1960,
     1104,1232,1360,1488,1616,1744,1872,2000
```

7.4 READING INFORMATION FROM THE CATALOG

The DOS command CATALOG provides very useful information, but unfortunately only in the screen, or in the printer (to get a hard copy of it, simply type PR#1 <return> and CATALOG). To delete a collection of programs, to rename them, or to handle them in any way, their names and related information (like the type of file, size, and whether they are locked or not) must be read and the command given directly from the keyboard, or from a program in the form of a string variable or constant.

Now, being able to read the screen, we will be able to read part of the catalog into string variables.

Program READ CATALOG initializes D$ in line 20, DIMensions CA$ ( ) to 24 (where the different lines of the catalog will be stored) and M% ( ) to 24 (the usual memory equivalents), and reads them. Line 40 performs a HOME and issues the command CATALOG.

At this point, if the entire catalog doesn't fit in the screen, a key must be pressed until the end is reached.

Line 50 starts the I and J loops, and line 60 stores into CA$ (I) the CHR$ of what is found in every location.

Line 70 produces a click after every line is read, to show the user that the computer is not dead, since no title can be placed on the screen, or it would come into CA$ ( ).

Lines 90 to 110 do a Bubble sort comparing the 33 rightmost characters, since to the left is the information about the type of file, etc., which would affect the sort and produce a different order.

In line 120 the last nonblank character is looked for, and the extra blanks disposed of. Line 150 PRINTs the catalog (the last 23 lines) in order.

```
1  REM  ** READ CATALOG **
20 D$ =  CHR$ (4):C = 1
30  DIM CA$(24),SP(24): FOR I = 1 TO 24: READ SP(
     I): NEXT : DATA  1024,1152,1280,1408,1536,16
     64,1792,1920,1064,1192,1320,1448,1576,1704,1
     832,1960,1104,1232,1360,1488,1616,1744,1872,
     2000
40  HOME : PRINT D$"CATALOG"
50  FOR I = 1 TO 24: FOR J = 0 TO 39
60 CA$(I) = CA$(I) +  CHR$ ( PEEK (SP(I) + J) - 128)
70  NEXT :X =  PEEK ( - 16336): NEXT
90  FOR I = 1 TO 23: FOR J = I + 1 TO 24
100  IF  RIGHT$ (CA$(I),33) >  RIGHT$ (CA$(J),33)
      THEN T$ = CA$(J):CA$(J) = CA$(I):CA$(I) = T$
110  NEXT : NEXT
120  FOR I = 1 TO 24: FOR J = 40 TO 1 STEP  - 1:X
     $ =  MID$ (CA$(I),J,1): IF X$ <  > " " AND X
     $ <  >  CHR$ (160) THEN CA$(I) =  LEFT$ (CA$
     (I),J): GOTO 140
130  NEXT J
140  NEXT I
142  FOR I = 1 TO 24:CA$(I) =  RIGHT$ (CA$(I), LEN
     (CA$(I)) - 7): NEXT
150  HOME : FOR I = 1 TO 24: PRINT CA$(I): NEXT
160  GET W$: HOME : END
```

7.5 VERY LOW RESOLUTION

Since there is no way to display graphics and text simultaneously, it is impossible, for example, to surround part of the screen with a frame, unless you use the methods explained in Section 9.15, which is not as easy as using text directly.

Program VERY LOW RESOLUTION lets you display 40 * 24 blocks on the text screen in white (no colors are available), which is exactly half the amount of blocks of low resolution. The good thing is they can appear on the screen with text of any kind.

Every block is actually a space PRINTed in INVERSE, which is equivalent to a white block the size of a character.

Two subroutines are available: Lines 1000 through 1040 act like the command HLIN, and lines 1100 through 1130 like VLIN.

In order to change to INVERSE, draw the necessary blocks and RETURN to the main program preserving the text mode active before the subroutines, the value of location 50 is stored in the variable MODE (position 50 holds the current Applesoft text mode), and POKEd back when the subroutines exit. See Table 7.1 for the values corresponding to the different modes.

Table 7.1. Text mode codes found in address 50.

VALUE	MODE
63	FLASH
127	INVERSE
255	NORMAL

All other values POKEd into location 50 make the PRINT command change some of the characters on the screen.

Line 10 clears the screen, sets FLASH mode, and prints "TEST" in a specific location of the screen. In order to show that the text mode is preserved, a frame is drawn while FLASH is (apparently) still active.

Lines 20 through 50 send in the corresponding variables the limits of the lines to draw.* When line 60 PRINTs "HELLO", it is still in FLASHing mode. Finally, line 70 PRINTs the 1000 first natural numbers in the same location, and the program terminates.

Apart from storing the mode, subroutines 1000 and 1100 are trivial.

```
1  REM  ** VERY LOW RESOLUTION **
10  HOME
15  FLASH : VTAB 10: HTAB 18: PRINT "TEST";
20 X1 = 10:X2 = 30:Y = 3: GOSUB 1000
30 X1 = 10:X2 = 30:Y = 20: GOSUB 1000
40 Y1 = 4:Y2 = 20:X = 10: GOSUB 1100
50 Y1 = 4:Y2 = 20:X = 30: GOSUB 1100
60  VTAB 17: HTAB 18: PRINT "HELLO";
70  NORMAL : FOR I = 1 TO 1000: VTAB 8: HTAB 18: PRINT
     I;: NEXT
99  CONT
1000  REM  ** HLIN X1,X2 AT Y **
1010 MODE =  PEEK (50)
1020  INVERSE
1030  FOR X = X1 TO X2: GOSUB 2000: NEXT
1040  POKE 50,MODE: RETURN
1100  REM  ** VLIN Y1,Y2 AT X **
1110 MODE =  PEEK (50)
1120  FOR Y = Y1 TO Y2: GOSUB 2000: NEXT
1130  POKE 50,MODE: RETURN
2000  HTAB X: VTAB Y: PRINT " ";: RETURN
```

---

* Subroutines 1000 and 1100 draw lines rather than blocks. If a single dot is needed, it can be drawn with, for example,

X1 = 5 : X2 = 5 : Y = 10 : GOSUB 1000,

which draws a block at locations X = 5 and Y = 10.

7.6 TEN CHARACTERS

Using the two subroutines above, program TEN CHARACTERS draws pseudo-graphic characters in the text screen. The lines to be drawn for each character are stored as DATA in lines 3000 through 3025. Only letters are available, but you can design your own numbers and symbols if you find it convenient.

Line 10 DIMensions LE% ( ) to store the numbers corresponding to the 26 letters, three components (Y1, Y2, and X), and as many groups of components as each letter needs (the letter with the biggest number of lines has seven), and M$ ( ) to store the letters on the screen.

Lines 15 through 40 READ the components of every line of every character; 333 is used as a separator between characters.

Line 110 starts the loop that will GET exactly 40 characters from the keyboard. If the character is a space, nothing needs to be drawn, and line 160 stores it in M$ ( ). Line 170 updates the cursor position (XC and YC are used, because X and Y are used by subroutine 2000).

If <esc> is typed, line 130 jumps to the menu, described later.

If the character is not a letter, line 140 ignores it and goes back to line 120. Otherwise, line 150 sets I to the ASCII code of the letter minus 65 (the lines corresponding to "A" are in LE% (0, ,)), and subroutine 900 is called.

Line 910 sets K to 0, to use it as the index of the lines to draw for character number I. Line 920 sets Y1, Y2, and X to the coordinates in LE% ( ), and calls subroutine 1100 to draw the line. K is incremented and the process is repeated until LE% (I, 0, K) is found to be 0, in which case the letter is completed.

The menu PRINTed by line 190 has three options:

1-DRAW
2-BSAVE
3-QUIT

1-DRAW

Line 220 calls subroutine 800. Line 810 clears the screen and sets XC and YC to 0, this program cursor's home. The L loop will get individual characters from M$ ( ) to draw them.

If M$ (L) is empty, there are no more characters, and the drawing is finished.

If M$ (L) is a space, no drawing is performed and line 850 updates the cursor. Otherwise, line 840 sends to subroutine 900 the code to draw the letter.

On return from subroutine 800, line 230 pauses until any character is pressed, before returning to the menu.

2-BSAVE

Line 300 asks for a filename and returns to the menu if answered with an empty string. Otherwise, it calls subroutine 800 to draw the entire screen, and BSAVEs page one of the text memory.

```
1  REM  ** TEN CHARACTERS **
10  HOME : DIM LE(26,2,10),M$(40):D$ =  CHR$ (4)
15  FOR I = 0 TO 25
20  READ Y: IF Y = 333 THEN 35
30 LE(I,0,K) = Y: READ LE(I,1,K): READ LE(I,2,K):
     K = K + 1: GOTO 20
35 K = 0
40  NEXT I
100  REM  ** GET CHARACTERS **
110  FOR L = 1 TO 40
120  GET W$: IF W$ = " " THEN 160
130  IF W$ =  CHR$ (27) THEN 190
140  IF W$ < "A" OR W$ > "Z" THEN 120
150 I =  ASC (W$) - 65: GOSUB 900
160 M$(L) = W$
170 XC = XC + 1: IF XC = 10 THEN XC = 0:YC = YC +
     1
180  NEXT L
190  HOME : VTAB 4: PRINT "1-DRAW": PRINT : PRINT
     "2-BSAVE": PRINT : PRINT "3-END ";
200  GET W$:V =  VAL (W$): IF V < 1 OR V > 3 THEN
     200
210  ON V GOTO 220,300,400
220  GOSUB 800
230  GET W$: GOTO 190
300  PRINT : INPUT "FILE NAME:";FL$: IF FL$ = "" THEN
     300
310 W$ =  LEFT$ (FL$,1): IF FL$ < "A" OR W$ > "Z"
      THEN 300
320  GOSUB 800
330  PRINT : PRINT D$"BSAVE "FL$",A1024,L1024"
340  GOTO 190
```

```
400  HOME : END
800  REM  ** DRAW THE ENTIRE SCREEN **
810  HOME :XC = 0:YC = 0: FOR L = 1 TO 40
820  IF M$(L) = "" THEN 870
830  IF M$(L) = " " THEN 850
840 I =  ASC (M$(L)) - 65: GOSUB 900
850 XC = XC + 1: IF XC = 10 THEN XC = 0:YC = YC + 1
860  NEXT
870  RETURN
900  REM  ** DRAW A LETTER **
910 K = 0
920 Y1 = LE(I,0,K) + YC * 6:Y2 = LE(I,1,K) + YC *
     6:X = XC * 4 + LE(I,2,K): GOSUB 1100
930 K = K + 1: IF LE(I,0,K) = 0 THEN  RETURN
940  GOTO 920
1000  REM  ** HLIN X1,X2 AT Y **
1010 MODE =  PEEK (50)
1020  INVERSE
1030  FOR X = X1 TO X2: GOSUB 2000: NEXT
1040  POKE 50,MODE: RETURN
1100  REM  ** VLIN Y1,Y2 AT X **
1110 MODE =  PEEK (50): INVERSE
1120  FOR Y = Y1 TO Y2: GOSUB 2000: NEXT
1130  POKE 50,MODE: RETURN
1200  REM  ** ERASE HLIN X1,X2 AT Y **
1210 MODE =  PEEK (50)
1220  NORMAL
1230  FOR X = X1 TO X2: GOSUB 2100: NEXT
1240  POKE 50,MODE: RETURN
1300  REM  ERASE VLIN Y1,Y2 AT X **
1310 MODE =  PEEK (50)
1320  FOR Y = Y1 TO Y2: GOSUB 2100: NEXT
1330  POKE 50,MODE: RETURN
2000  HTAB X: VTAB Y: PRINT " ";: RETURN
3000  DATA   2,5,1,1,1,2,2,5,3,4,4,2,333
3001  DATA  1,5,1,1,1,2,3,3,2,5,5,2,1,5,3,333
3002  DATA  1,5,1,1,1,2,5,5,2,1,1,3,5,5,3,333
3003  DATA  1,5,1,1,1,2,5,5,2,2,4,3,333
3004  DATA 1,5,1,1,1,2,3,3,2,5,5,2,1,1,3,5,5,3,333
3005  DATA  1,5,1,1,1,2,3,3,2,1,1,3,333
3006  DATA  1,5,1,1,1,2,5,5,2,1,1,3,4,5,3,333
3007  DATA  1,5,1,3,3,2,1,5,3,333
3008  DATA  1,5,2,333
3009  DATA  4,4,1,5,5,2,1,5,3,333
3010  DATA  1,5,1,2,4,2,1,2,3,4,5,3,333
3011  DATA  1,5,1,5,5,2,5,5,3,333
3012  DATA  1,5,1,2,2,2,1,5,3,333
3013  DATA  1,5,1,2,2,2,2,5,3,333
3014  DATA  1,5,1,1,1,2,5,5,2,1,5,3,333
3015  DATA  1,5,1,1,1,2,3,3,2,1,3,3,333
3016  DATA  1,4,1,1,1,2,4,4,2,1,5,3,333
3017  DATA  1,5,1,1,1,2,3,4,2,1,3,3,5,5,3,333
```

```
3018  DATA  1,3,1,5,5,1,1,1,2,3,3,2,5,5,2,1,1,3,3
     ,5,3,333
3019  DATA  1,1,1,1,5,2,1,1,3,333
3020  DATA  1,5,1,5,5,2,1,5,3,333
3021  DATA  1,4,1,4,5,2,1,4,3,333
3022  DATA  1,5,1,4,4,2,1,5,3,333
3023  DATA  1,2,1,4,5,1,3,3,2,1,2,3,4,5,3,333
3024  DATA  1,3,1,3,5,2,1,3,3,333
3025  DATA  1,1,1,4,5,1,1,1,2,3,3,2,5,5,2,1,2,3,5
     ,5,3,333
```

7.7 USING THE SECOND PAGE OF TEXT

Just as in high resolution, there are two pages in the TEXT/LOW RESOLUTION GRAPHICS modes. Unfortunately, there is no way to write directly to page two.

One way of doing so is to use the memory addresses of page one, add the distance between the two pages, and POKE into page two the contents of page one (in program scroll every line was being moved one place; this new program would move every character exactly 1024 positions). Unfortunately it is a slow process and the only convenience would be to have a copy of page one.

Usually page two of text is used to store the beginning of the program. If page two is used in any way, the program is destroyed. Therefore, to be able to use page two of text, the program must be relocated (see 11.7).

A better way to put information in page two is BLOADing a file that originated in page one and therefore is a perfect copy of the screen. If the address of the second page is specified (2048 in decimal or $800 in hexadecimal) when BLOADing, the screen is placed in the second page.

To display the second page, we use the same soft switches as in high resolution. POKE −16300, 0 displays page one and POKE −16299, 0 shows page two.

7.8 HELP FILES

When a program is so complex that it is not easy to remember all its commands and options, it is convenient to have a HELP option in every menu, containing the information needed to use correctly that particular part of the program.

Unfortunately, having a good documentation within the program takes too much memory.

A good solution is to keep all the text screens already BSAVEd so that a BLOAD brings them into memory immediately. The process of BLOADing a file into the text screen is relatively fast but nevertheless is disturbing when the page being BLOADed is the same one being displayed.

It is a much better idea to BLOAD one page while the other one is being shown and, when the process is finished, the pages are flipped.

The only way to do this using only one page of text is to switch temporarily to a blank high resolution page, which is certainly not equivalent.

To use program HELP you must first run program RELOCATE (see 11.7) using HELP as the program name and 3072 as address.

Program HELP uses four BSCREENS that you must create (using the editor, for example). In line 20 it displays a fake menu (the only option that works is the third one) in which a HELP option is offered.

If that option is selected, line 10000 sets H to 0 (there are four Help files in this program, and H is going to indicate which one to load) and FL$ to "8". Since the memory addresses of the pages are (in hexadecimal) $400 and $800, this FL$ will contain either a "4" or an "8", to place the screen into the desired text page.

Line 10010 increments H. If it is already 5, the program jumps to line 10025, to avoid loading a non-existent fifth BSCREEN.

Line 1020 BLOADs the corresponding file: for convenience, they have been called HELP1, HELP2, HELP3, and HELP4. That way, H (which keeps track of the file to load) can be used as an index to load the appropriate one. Two 0's are added to FL$ to complete the address.

The first screen BLOADed is placed in the second page (since FL$="8"), while page one is displayed. As soon as it is ready in memory, line 1022 flips the pages by POKEing 0 into −16299 (show page two) and jumps to line 10030 where FL$ is toggled. Line 10050 jumps to 10010 to increment H and see if this is the last page. If there are more screens to load, the new page is loaded into the page that is not being displayed, and line 10027 waits until the user presses either <esc> (back to the menu displaying page one), or any other key (next page).

This way, the screen is not affected while the next one is loaded, smoothing operations and giving the user time to read the instructions.

```
1   REM   ** HELP **
10 D$ =  CHR$ (4)
20  HOME : VTAB 8: PRINT "1-OPTION 1": PRINT : PRINT
     "2-OPTION 2": PRINT : PRINT "3-HELP": PRINT
     : PRINT "4-QUIT ";
30  GET W$: IF W$ = "3" THEN 10000
40  PRINT : PRINT : PRINT "OPTION "W$" SELECTED":
      END
10000 H = 0:FL$ = "8"
10010 H = H + 1: IF H = 5 THEN 10025
10020  VTAB 1: PRINT : PRINT D$"BLOAD HELP" STR$
     (H)",A$"FL$"00"
10022  IF H = 1 THEN  POKE  - 16299,0: GOTO 10030
10025  VTAB 24: HTAB 40: IF FL$ = "4" THEN  POKE
     3063,96
10027  GET W$: IF H = 5 THEN 20
10028  IF W$ =  CHR$ (27) THEN  POKE  - 16300,0: GOTO
     20
10030  IF FL$ = "4" THEN  POKE  - 16300,0:FL$ = "
     8": GOTO 10050
10040  POKE  - 16299,0:FL$ = "4"
10050  GOTO 10010
```

7.9 MORE THAN TWO PAGES IN TEXT

If the program is relocated to a high address, new pages can be defined in locations $1200 hexadecimal (3072 decimal) for page three, $1600 (4096) for page four, etc. There is no way to display these pictures directly, but they can be used to store temporarily text or graphics that can be transferred to pages one or two using either the BSAVE-BLOAD options or the memory addresses (to transfer page three to page one, for example, use the usual I and J loops with a POKE M% (I)+J, PEEK (M% (2048) +J)).

Since the only difference between text and low resolution graphics is the state of one soft switch, everything said about the two and more pages applies to low resolution.

7.10 TEXT EDITOR

One of the most useful tools when programming is an editor, in this case to free you from all the HTAB's and VTAB's it takes to arrange a screen.

The complexity of an editor can be limitless. Here we will include the most frequently used features, and we will concern ourselves with only 24 lines of 40 characters, enough for the screen of the Apple.

Since this is a program of relative complexity, a user's instructions section is presented here before the actual explanation.

User's Instructions

When the first option of the initial menu (EDIT) is selected, the screen goes blank and displays the cursor at the home position (HTAB 1, VTAB 1). Any text entered in that line will be recorded and stored in that position.

The left arrow behaves differently here than it does in Applesoft, because it erases the characters as it backspaces; the right arrow is not an accepted character and has no effect.

Trying to type more than 40 characters makes the speaker beep, and the characters are discarded.

When there are no characters on the line, the left arrow beeps too.

<return> and ctrl-A are used to go to the next line and to terminate the input of a line.

ctrl-F moves the cursor up.

When something is typed over a nonempty line, the previous contents are lost and replaced by the new text. However, if you want to keep the previous line and discard the new characters, use the left arrow to delete all the new lines. The old one is thus recuperated, even though it is not visible immediately.

ctrl-V adds one character to the left of the line where the cursor lies, therefore allowing easy centering and positioning of titles.

ctrl-T deletes the first character of the line where the cursor lies.

ctrl-shift-N adds one line before the one where the cursor lies. The bottom line disappears, but can still be recuperated (see ctrl-shift-m).

ctrl-shift-M deletes one line at the cursor position, bringing back at the bottom the line deleted with ctrl-shift-n. Only one line is remembered.

ctrl-E enters the edit mode, which gives access to the following subcommands:

Space - Moves the cursor without deleting the characters it passes over.

Left arrow - Moves the cursor to the left nondestructively.

Ctrl-I - Enters insert mode, indicated at the edit window; any character typed while in this mode is inserted where the cursor lies, as long as there is space. To exit insert mode, press <esc>.

Ctrl-D - Deletes the character at the cursor position.

Ctrl-C - Enters change mode. Any character typed at the cursor position will replace the one previously there. To exit change mode, type <esc>.

Ctrl-R - Recuperates the current line as it was before any editing was done.

<return> - Leaves edit mode.

When a file is created by this program, it contains the 24 lines as strings, even if some (or all) are empty. Therefore, to be able to load a file it must have at least 24 lines.

The CLEAR SCREEN option deletes everything from the screen.

Line 10 initializes some variables: T$ ( ) will store the text; M% ( ) will be used to store the addresses of the first column of the video memory; PR$ ( ) will store the program lines before they are PRINTed to the EXEC file; L will keep the length of the current line; BS$ will store a backspace, a space, and another backspace. When BS$ is printed, it erases the character to the left of the cursor and leaves it there.

X and Y will keep the cursor position, TRUE and FALSE their boolean equivalents, Q$ a quote, and LF$ a line feed.

Line 100 shows the menu, with the following options:

1-EDIT
2-CLEAR SCREEN
3-SAVE
4-LOAD
5-CREATE PROGRAM
6-PRINT
7-QUIT

1-EDIT

When this option is selected, line 755 stores in the variable T the current value of Y (the vertical line), and sets FL to FALSE.

Line 760 call subroutine 2000, which PRINTs the entire T$ ( ). The first 23 lines are PRINTed as they are in T$ ( ), calling subroutine 10000 (sets the cursor at X, Y). If the length of T$ (24) is smaller than 40, line 2020 PRINTs it; otherwise, the first 39 characters are PRINTed, and the last one (character # 40) is POKEd directly into its corresponding position in memory, since PRINTing it in the rightmost, bottom position of the screen produces a scroll, erasing the topmost line.

Before returning from subroutine 2000, Y is set to T, which keeps the value of Y before the screen was drawn, therefore leaving the cursor where it was before. Once the screen is ready, line 760 transfers control to line 510.

The GET fetches any character coming from the keyboard. If it is a displayable character (between " " and "↑"), it jumps to line 1550. This comparison is made here, because entry of characters, not of commands, is the most frequent situation and it will reduce the delay in the actual input of text.

Line 1550 doesn't accept the character if FL is true. Line 1560 rejects the character if the current line already has L characters (L is 40 for lines 1-23 and 39 in line 24, to treat the special case of the last character of the screen which causes an unwanted scroll). Otherwise, line 1570 accepts it, PRINTs it on the screen, and adds it to X$, the temporary buffer where the line currently being edited is kept.

By storing this line in X$ without touching the place where it will be definitely stored (T$ (Y)), the program can recuperate the line as it was before the alterations made are permanently transferred to T$ (Y).

If the character typed was a left arrow (CHR$ (8)), lines 530 through 550 deal with it in the same way as for the controlled input (see chapter 1).

If the character typed was a <return> or a ctrl-A (they behave as the same character in this program), line 600 calls subroutine 9500, which RETURNs if X$ is empty (nothing has been typed and the current line needs no change), or assigns X$ to T$ (Y), empties X$, and clears to the end of the line if the length is less than 40.

On return from subroutine 9000, Y is incremented (the cursor jumps to the next line) if Y is not already in the bottom of the screen (line 24). Subroutine 9000 sets L to 39 if Y is 24 (again, to prevent the scroll), and subroutine 10000 puts the cursor where it belongs.

If the character typed is ctrl-F (CHR$ (6)), line 625 calls subroutine 9500 to transfer X$ to T$ (Y) if needed, and decreases Y if it is not in the top line already. Subroutine 9000 now performs the opposite than when called from line 610: if the cursor was in line 24, it sets L back to 40. The cursor is set in position.

If W$ is equal to ctrl-T (CHR$ (20)), line 660 calls the usual subroutine 9500 and nothing is done if no characters are to be moved. Otherwise, line 670 takes care of the case in which there is only one character in T$ (Y). This is to prevent the error when trying to evaluate LEFT$ (T$ (Y), LEN (T$ (Y)), since the position parameter of LEFT$ cannot be 0. T$ (Y) is set to empty, and the character is deleted from the screen. Line 680 trims off the left character of T$ (Y), and PRINTs the resulting string.

If W$ is <esc>, line 690 jumps to the menu, without updating the last line. Therefore, going to the menu and returning to the edit option recuperates a line that might have been started, and deleted with the left arrow.

If W$ is ctrl-V (CHR$ (22)), lines 715 to 730 add one blank to the left of T$ (Y) if its length is not equal to L, in a way similar but opposite to ctrl-T.

If W$ = ctrl-shift-N, lines 750 to 760 move every line under the cursor one place down, and empty T$ (Y). By using T$ (25), the last line (T$ (24)) is not lost, and can be recuperated.

If W$ = ctrl-shift-M, lines 800 to 820 delete the line where the cursor is and shift every row one place up, bringing back line 25. Line 815 takes care of the anomalous case of the 40th character of T$ (24).

If W$ = ctrl-E (CHR$ (5) or enter edit mode), line 840 enters a whole new character entry cycle. After all the previous commands finish their function, they return control to line 510, the main input circle. In the edit mode, some characters will have different meanings than outside it, and therefore this one will be a new cycle that will be exited from only by pressing <return>.

Line 840 calls the usual 9500, and sets NY to either 1 or 10, depending on Y. Line 860 calls subroutine 8400, which PRINTs in INVERSE mode the title EDIT ON, using NY as the VTAB. If the line to be edited is 1 and NY were set to 1, this title would obstruct the vision.

Line 860 makes a copy of T$ (Y). All the alterations will be made on X$, and only when exiting the edit mode will T$ (Y) be updated. This permits recuperation of the original line after changes.

Line 870 GETs W$; if it is a space, line 880 rejects it if the cursor is at either the last character of X$ or at one position to the right of L. If it is accepted, line 900 increases X and calls subroutine 10000, therefore moving the cursor one place to the right without affecting X$. This is the nondestructive cursor movement.

If W$ = ctrl-D (CHR$ (4) or delete one character), line 920 ignores the command if either X$ is empty or if the cursor is after the end of the string. If there is only one character in X$, line 930 deletes it. Subroutine 8500 PRINTs the line, taking care of the by-now famous last character of the screen. Lines 940 and 950 take care of the cases in which the cursor is at the first or last position of X$, to avoid evaluating RIGHT$ or LEFT$ with a 0 parameter. Finally, line 960 treats the normal case, deleting the character where the cursor lies.

If W$ = backspace (CHR$ (8) or nondestructive cursor left), line 980 ignores the command if it is already at the left of the screen. Otherwise, list 990 decrements X by 1 and calls subroutine 10000 which sets the cursor in X, Y.

If W$ = ctrl-I (CHR$ (9) or enter insert mode), line 1010 PRINTs INSERT ON in the edit window. Line 1020 GETs one character; if it is <esc>, it calls subroutine 8400 (change INSERT ON for EDIT ON in the edit window) and exits insert mode. Otherwise, line 1030 rejects any nonprintable character. Line 1035 rings the bell if X$ has as many characters as the value of L. Line 1040 treats the case in which the cursor is at the first position of

X$. Line 1050 treats the one in which X is after the last character of X$, and line 1060 treats the normal case. In all three, W$ is added to X$ in the place where the cursor lies and subroutine 8500 PRINTs the new X$.

If W$ = ctrl-C (CHR$ (3) or enter change mode), line 1080 rejects the command if there is nothing to change in X$. Line 1090 PRINTs CHANGE ON in the edit window. Line 1100 GETs a character and leaves the change mode if it is <esc>. Line 1110 rejects the character if it is nonprintable or if the cursor is not above a character, but after the end of X$. Lines 1120, 1130, and 1140 handle the cases in which there is only one character, or the character is at the extreme left or right of X$. Line 1050 treats the normal case, when the character is in the middle of X$.

In all these lines subroutine 8500 is called to draw the new line.

If W$ = ctrl-R (CHR$ (18) or recuperate line before changes), line 1170 sets X$ = T$ (Y) forgetting all the changes made. The new X$ is PRINTed by subroutine 8500, and if the cursor is beyond the last character of this new line, it is brought back to one place after the end of X$.

If W$ = <return> (CHR$ (13) or ctrl-M or leave edit mode), line 1200 sets T$ (Y) = X$, temporarily keeping the line position in T to use Y to erase the edit window. List 1210 puts Y and the cursor where they were previously, X$ is emptied, and the program goes back to the regular input routine (line 510), leaving the edit mode.

2-CLEAR SCREEN

Line 3000 asks for confirmation, to prevent an accidental mistake; if the choice is confirmed, line 3020 sets all T$ ( ) to " ".

3-SAVE

Line 200 shows a new menu:

1-BSAVE
2-AS TEXT
3-MENU

1-BSAVE. Line 220 sets F1 to TRUE and calls subroutine 2300 for a filename. For the text files, line 2320 OPENs it; since this is a binary file, F1 makes the subroutine RETURN before it is OPENed.

Subroutine 2000 PRINTs the entire T$ ( ); VTAB 1 sends the cursor to the first line, because the last line of the program was PRINTed followed by a semicolon. If the D$ of the disk command is PRINTed at that moment, it will not have any effect, and if the cursor is at the bottom line and a PRINT is executed, a scroll will be produced. Moving the cursor to the first line, PRINT does not have any visible effect (since there is no cursor present at the moment).

The file is BSAVEd.

2-AS TEXT. Line 240 shows two new options:

1-TO BE USED AS AN EXEC FILE
2-TO BE READ BY THIS PROGRAM

The only difference between the two is that when the file created will be used as an EXEC file, no number can be used at the beginning of the file to indicate how many lines are being PRINTed, as is the case with second option.

To indicate that the first option has been chosen, line 242 sets FL to TRUE.

Line 244 calls subroutine 2300, which prompts for a filename. If <return> is pressed, the subroutine POPs (stops being a subroutine and becomes a regular instruction) and goes back to the menu. Line 2310 rejects the name if it doesn't begin with a character from "A" to "Z", since it isn't a valid name for a file. List 2320 OPENs the file, since this subroutine will be used by the SAVE and LOAD options. Line 250 PRINTs T$ ( ) to the disk, CLOSEs the file, and returns to the menu.

4-LOAD

Line 400 shows a new menu:

1-AS BINARY
2-AS TEXT
3-MENU

1-AS BINARY. Line 444 sets F1 to TRUE, and calls subroutine 2300 for a filename; the file is BLOADed.

Lines 445 through 455 scan every line and row of the video screen (using M% ( ) for the addresses of the lines), and puts the characters in T$ ( ). A click is emitted after each line is analyzed.

When the whole screen has been processed, line 455 calls subroutine 2200, which deletes all the blank characters to the right of the last nonblank of every line in T$ ( ), and sets N equal to the index of the last nonempty line.

2-AS TEXT. Line 404 shows the two options:

1-CREATED BY THIS PROGRAM
2-CREATED OTHERWISE

As with the SAVE option, the difference between the two options is that one has the number of lines to read at the beginning of the file. However, if that number is unknown, there is a great chance of causing an END OF DATA error.

To prevent this, when option 2 is selected, line 407 turns on an ONERR GOTO routine that will CLOSE the file and continue normally when the end of the file is reached before 24 lines are read.

Line 408 calls subroutine 2300 for a filename, and sets it to READ.

Line 410 empties T$ ( ). Line 411 reads N if FL indicates that the file contains this number before the actual text. If there is no N to read, 24 is assumed in line 412.

Line 413 starts the I loop that will read all the lines into T$ ( ).

Line 415 reads the characters one by one using the GET statement instead of INPUT, so that no characters are ignored. The trailing blanks are considered, the commas and colons are read normally, and the quotes are treated as any other character.

If the character read is a <return> or a line feed (CHR$ (10)), line 415 jumps to line 427, incrementing I, therefore terminating the current line.

If the character is outside the upper case character range, line 418 ignores it.

If the length of T$ (I) has already reached 40 characters, W$ is totally ignored, as are all the characters following it, until a <return> or line feed is encountered.

Otherwise, line 420 adds W$ to T$ (I), and the next character is read.

If the second option (CREATED OTHERWISE) is selected and the file ends before the 24 lines are read, line 430 resets the ONERR GOTO by executing a POKE 216, 0, CLOSEs the file, and goes back to the menu.

Otherwise, the file is CLOSEd normally and the ONERR GOTO is disabled.

5-CREATE PROGRAM

This option is basically a DATA CREATOR (see section 5.2) that is going to use PRINT commands instead of storing the information as DATA. It has two special cases to consider:

1. When the character is a quote, it cannot be included in the string. Instead, the preceding ?''..... must be closed with a quote, and the real one added, either using the CHR$ (34) or its equivalent in a string variable.
2. If the last character of the screen is used, it should not be included in the string to be PRINTed, because that would cause a scroll that would erase the first line. Instead, an instruction should be used to POKE into the corresponding address of the video memory (2039) the number that will produce the character wanted.

Line 6000 asks for an initial line number, and rejects it if it is not valid. Line 6005 asks for the increment to use, and line 6006 PRINTs the title ''PROCESSING.''

Line 6010 initializes C to 1 (the index of the line of PR$ ( ) currently being processed). IL gets the initial line number. PR$ (1) gets the STR$ of the initial line number, plus 'HOME:?'' ' (apostrophes are used here to delimit the string to show that the quote is actually included). Finally, LN is incremented IN units.

Line 6020 calls subroutine 2200, which deletes all the redundant blanks at the right hand side of every T$ ( ), and adjusts N to the last nonempty line; If N is 0, there is no program to create and the program jumps back to the menu.

Line 6025 begins the I loop that will process every line in T$ ( ). If the current line is empty (its LENGTH is 0), the only thing needed is a carriage return (which sends the cursor to the left margin of next line), and since LF$ was set to CHR$ (13), it is added to the current PR$ ( ) by means of subroutine 7000, explained later, and the program is branched to line 6040, where the I loop goes to the next cycle.

If the line was not found empty, line 6030 starts a J loop that will study every character of T$ (I).

If the character under study is the last of the screen, lines 6032 through 6035 end the string in the PR$ (I) last processed by adding a quote, add a semicolon, and add the instruction POKE 2039, followed by the ASCII code that, when POKEd, will correspond to the character in W$.

If the position is not the last one of the screen, line 6037 puts in X$ the Jth character of T$ (I), and sends it to subroutine 7000, which is in charge of arranging PR$ ( ). At the end of the J loop, if the length of T$ (I) is found to be less than 40, a <return> character is added, so that the next T$ (I) starts in a new line; when the line has exactly 40 characters, the following line can be PRINTed as a continuation of the current one.

When the I loop is finished, in line 6040, if FL is TRUE there has been at least one character added to PR$ (C), a quote and a semicolon are added to PR$ (C). Otherwise, line 6050 decrements C by one, forgetting completely the last PR$ ( ).

Lines 6060 through 6080 produce the file shown in Program Fragment 7.1.

```
32000 J=1: DIM M% (2000): D$=CHR$ (4): ?D$"OP
EN DUMMY1":?D$"READ DUMMY 1":INPUT AC: FOR
 I=1 TO AC: INPUT M% (I): NEXT: ?D$"CLOSE"
32001 FOR I=2054 TO 10000: IF PEEK (I)=42
THEN POKE I, M% (J): AC=AC-1:J=J+1: GOTO 320
03
32003 NEXT
32002 IF AC=0 THEN DEL 32000, 32004
100 HOME:?"**********************";
RUN 32000
```

Program Fragment 7.1

Lines 6087 through 6140 PRINT PR$ ( ) to the file DUMMY, which is still open, in a way similar to the one used in program HIDDEN. The regular characters will be PRINTed as asterisks, to let PROGRAM FRAGMENT X (when EXECed) POKE directly the actual characters into those positions in memory. The Q$'s and separators (semicolons, colons, line numbers, and quotes) are left intact.

Since the quotes are the main indicator of the end of a string and the beginning of a sequence of one or more characters that actually must be PRINTed and not replaced by asterisks, when line 6090 finds one quote, it toggles F2, which is used in line 6110 to jump to either one of the two states: PRINT the character, or PRINT an asterisk. AC is used as a counter to track the number of asterisks PRINTed.

At the end of the loops, the instruction RUN 32000 is added and the file is CLOSEd.

If the character in X$ is not a quote, line 7000 jumps to 7010. FL is set to TRUE, indicating that the current line (PR$ (C)) has at least one character. X$ is added to PR$ (C), and if PR$ (C) is found to be 235 characters long, or longer, subroutine 7100 is called. To end a line, the string (if any) must be closed with a quote, and a semicolon added. Yet, if the last character found is a quote, PR$ (C) would look like this:

```
(more characters)      ........"Q$"
```

and line 7100 deletes the last quote. Otherwise, the quote needed to close the string is added in line 7110, and line 7120 adds the semicolon.

On return from subroutine 7100, line 7010 sets FL to FALSE, to indicate that the next line is empty. C is incremented, and the STR$ of the line number, a "?" (for PRINT), and a quote are added to PR$ (C).

If the character in X$ is a quote, line 7003 sets F1 to TRUE (explained later). If the line is already so long that it cannot take the extra character that must be added, subroutine 7100 is called (explained previously), C is increased, and the STR$ of the line number, a Q$ (which is going to be a quote in the final version of the program), and a quote are added to the new PR$ (C).

If the last character sent to subroutine 7000 was a quote, PR$ (C) looks like this:

```
(characters) ....."Q$"
```

and since another Q$ must be added, it would be a waste to close the quotes and add Q$. Instead, the last quote is deleted and Q$" is added to PR$ (C).

If the quote must be added to a string not included in the two last cases, ' "Q$" ' is added to PR$ (C) (the apostrophes are used here to denote that the quotes are within the actual string).

Line 6160 OPENs the file DUMMY1, where the ASCII code of the characters that had to be PRINTed as asterisks will be stored. Unlike DUMMY, it does not matter if this file ends up having some garbage at the end, because it is going to be EXECed.

AC (the number of asterisks PRINTed to DUMMY) is PRINTed, and using F2 to indicate whether the following character is inside a pair of quotes or not, lines 6170 through 6210 PRINT the ASCII code of only those characters that were replaced by asterisks in DUMMY.

Line 6220 PRINTs a message instructing how to EXEC DUMMY.

6-PRINT

Line 3600 asks for a left margin (the amount of characters to insert to the left of every T$ (I), so that the horizontal position can be varied). Lines 3610 through 3625 allow a return to the menu. Line 3630 sets the output to be sent to the printer, and starts the I-loop. Line 3640 jumps to line 3660 if no left margin is required (remember a FOR loop is always executed at least once). Line 3650 PRINTs as many blanks as asked for, and line 3660 PRINTs the actual T$ ( ). Line 3670 returns the output to the screen and returns to the menu.

```
1   REM  ** EDITOR **
10   DIM T$(25),M%(24),PR$(10):L = 40:Y = 1:BL$ =
       CHR$ (7):BS$ =  CHR$ (8) + " " +  CHR$ (8):
      X = 1:Y = 1:D$ =  CHR$ (4):TRUE = 1:FALSE =
      0:Q$ =  CHR$ (34):LF$ =  CHR$ (13)
20   FOR I = 1 TO 24: READ M%(I): NEXT : DATA  102
      4,1152,1280,1408,1536,1664,1792,1920,1064,11
      92,1320,1448,1576,1704,1832,1960,1104,1232,1
      360,1488,1616,1744,1872,2000
100   HOME : VTAB 6: PRINT "1-EDIT": PRINT : PRINT
      "2-CLEAR SCREEN": PRINT : PRINT "3-SAVE": PRINT
      : PRINT "4-LOAD"
110   PRINT : PRINT "5-CREATE PROGRAM": PRINT : PRINT
      "6-PRINT": PRINT : PRINT "7-QUIT ";
120   GET W$: IF W$ < "1" OR W$ > "7" THEN 120
```

```
130  ON  VAL (W$) GOTO 755,3000,200,400,6000,3600
     ,3500
200  HOME : VTAB 9: PRINT "1-BSAVE": PRINT : PRINT
     "2-AS TEXT": PRINT : PRINT "3-MENU ";
205  GET W$:: IF W$ < "1" OR W$ > "3" THEN 205
210  ON  VAL (W$) GOTO 220,240,100
220 F1 = TRUE: GOSUB 2300: GOSUB 2000: VTAB 1: PRINT
     : PRINT D$"BSAVE "FL$",A$400,L$400": GOTO 20
     0
240  HOME : VTAB 10: PRINT "1-TO BE USED AS AN EX
     EC FILE": PRINT : PRINT "2-TO BE READ BY THI
     S PROGRAM ";
241  GET W$: IF W$ = "1" THEN FL = FALSE: GOTO 24
     4
242  IF W$ = "2" THEN FL = TRUE: GOTO 244
243  GOTO 241
244  GOSUB 2200:F1 = FALSE: GOSUB 2300: PRINT D$"
     WRITE "FL$
245  IF FL THEN FL = FALSE: PRINT N
250  FOR I = 1 TO N: PRINT T$(I): NEXT : PRINT D$
     "CLOSE": GOTO 200
400  HOME : VTAB 8: PRINT "1-AS BINARY": PRINT : PRINT
     "2-AS TEXT": PRINT : PRINT "3-MENU ";
401  GET W$: IF W$ < "1" OR W$ > "3" THEN 400
402  ON  VAL (W$) GOTO 444,404,100
404  HOME : VTAB 10: PRINT "1-CREATED BY THIS PRO
     GRAM": PRINT : PRINT "2-CREATED OTHERWISE ";

405  GET W$: IF W$ = "1" THEN FL = TRUE: GOTO 408

406  IF W$ = "2" THEN FL = FALSE: GOTO 408
407  GOTO 405
408  GOSUB 2300: PRINT D$"READ "FL$
410  FOR I = 1 TO 24:T$(I) = "": NEXT
411  IF FL THEN  INPUT N: GOTO 413
412 N = 24
413  FOR I = 1 TO N
415  GET W$: IF W$ =  CHR$ (13) OR W$ =  CHR$ (10
     ) THEN 427
418  IF W$ < " " OR W$ > "^" THEN 415
419  IF  LEN (T$(I)) = 40 THEN 415
420 T$(I) = T$(I) + W$
425  GOTO 415
427  NEXT
430  POKE 216,0: PRINT : PRINT D$"CLOSE": GOTO 10
     0
440  IF  PEEK (222) = 5 THEN 430
442  POKE 216,0: RESUME
444 F1 = TRUE: GOSUB 2300: PRINT D$"BLOAD "FL$
445  FOR I = 1 TO 24:T$(I) = "": FOR J = 0 TO 39
450 W =  PEEK (M%(I) + J) - 128: IF W < 32 THEN 4
     55
```

```
452 T$(I) = T$(I) +  CHR$ (W)
455  NEXT :W =  PEEK ( - 16336): NEXT : GOSUB 220
     0: GOTO 100
500 X$ = ""
510  GET W$: IF W$ >  = " " AND W$ <  = "^" THEN
     1550
520  IF W$ <  >  CHR$ (8) THEN 560
530  IF X$ = "" THEN  PRINT BL$;: GOTO 510
540  IF  LEN (X$) = 1 THEN X$ = "": PRINT BS$;: GOTO
     510
550 X$ =  LEFT$ (X$, LEN (X$) - 1): PRINT BS$;: GOTO
     510
560  IF W$ <  >  CHR$ (13) AND W$ <  >  CHR$ (1) THEN
     620
600  GOSUB 9500: IF Y = 24 THEN 510
610 Y = Y + 1: GOSUB 9000: GOSUB 10000: GOTO 510
620  IF W$ <  >  CHR$ (6) THEN 650
625  GOSUB 9500
630  IF Y = 1 THEN 510
640 Y = Y - 1: GOSUB 9000: GOSUB 10000: GOTO 510
650  IF W$ <  >  CHR$ (20) THEN 690
660  GOSUB 9500: IF T$(Y) = "" THEN 510
670  IF  LEN (T$(Y)) = 1 THEN T$(Y) = "": GOSUB 1
     0000: PRINT " "BS$;: GOTO 510
680 T$(Y) =  RIGHT$ (T$(Y), LEN (T$(Y)) - 1): GOSUB
     10000: PRINT T$(Y)" ";: GOSUB 10000: GOTO 51
     0
690  IF W$ =  CHR$ (27) THEN X$ = "": GOTO 100
710  IF W$ <  >  CHR$ (22) THEN 740
715  GOSUB 9500: IF X$ <  > "" THEN T$(Y) = X$:X$
      = ""
720  IF  LEN (T$(Y)) = L THEN 510
730 T$(Y) = " " + T$(Y): GOSUB 10000: PRINT T$(Y)
     ;: GOSUB 10000: GOTO 510
740  IF W$ <  >  CHR$ (30) THEN 790
750  GOSUB 9500: FOR I = 25 TO Y + 1 STEP  - 1:
     T$(I) = T$(I - 1): NEXT :T$(Y) = ""
755 T = Y:FL = FALSE
760  GOSUB 2000: GOTO 510
790  IF W$ <  >  CHR$ (29) THEN 830
800 T = Y: FOR I = Y TO 24:T$(I) = T$(I + 1): NEXT
     :T$(25) = ""
810  FOR Y = 1 TO 23: GOSUB 10000: PRINT T$(Y);
     : CALL  - 868: NEXT : GOSUB 10000: IF  LEN
     (T$(24)) < 40 THEN  PRINT T$(24);: CALL  -
     868: GOTO 820
815  PRINT  LEFT$ (T$(24),39);: POKE 2039, ASC
     ( RIGHT$ (T$(24),1)) + 128
820 Y = T: GOSUB 10000: GOTO 510
830  IF W$ <  >  CHR$ (5) THEN 510
840  GOSUB 9500: IF Y = 1 THEN NY = 10: GOTO 86
     0
```

```
850 NY = 1
860  GOSUB 8400:X$ = T$(Y)
870  GET W$: IF W$ <  > " " THEN 910
880  IF X = L + 1 OR X =  LEN (X$) + 1 THEN 870

900 X = X + 1: GOSUB 10000: GOTO 870
910  IF W$ <  >  CHR$ (4) THEN 970
920  IF X$ = "" OR X =  LEN (X$) + 1 THEN 870
930  IF  LEN (X$) = 1 THEN X$ = "":X = 1: GOSUB
     8500: GOTO 870
940  IF X = 1 THEN X$ =  RIGHT$ (X$, LEN (X$) -
     1): GOSUB 8500: GOTO 870
950  IF X =  LEN (X$) THEN X$ =  LEFT$ (X$, LEN
     (X$) - 1): GOSUB 8500: GOTO 870
960 X$ =  LEFT$ (X$,X - 1) +  RIGHT$ (X$, LEN (
     X$) - X): GOSUB 8500: GOTO 870
970  IF W$ <  >  CHR$ (8) THEN 1000
980  IF X = 1 THEN 870
990 X = X - 1: GOSUB 10000: GOTO 870
1000  IF W$ <  >  CHR$ (9) THEN 1070
1010  HTAB 16: VTAB NY: INVERSE : PRINT "INSERT
     :ON";: NORMAL : GOSUB 10000
1020  GET W$: IF W$ =  CHR$ (27) THEN  GOSUB 84
     00: GOTO 870
1030  IF W$ < " " OR W$ > "^" THEN 1020
1035  IF  LEN (X$) >  = L THEN  PRINT BL$;: GOTO
     1020
1040  IF X = 1 THEN X$ = W$ + X$: GOSUB 8500:X =
     X + 1: GOSUB 10000: GOTO 1020
1050  IF X =  LEN (X$) + 1 THEN X$ = X$ + W$: GOSUB
     8500:X = X + 1: GOSUB 10000: GOTO 1020
1060 X$ =  LEFT$ (X$,X - 1) + W$ +  RIGHT$ (X$,
      LEN (X$) - X + 1): GOSUB 8500:X = X + 1: GOSUB
     10000: GOTO 1020
1070  IF W$ <  >  CHR$ (3) THEN 1160
1080  IF X$ = "" THEN 870
1090  HTAB 16: VTAB NY: INVERSE : PRINT "CHANGE:O
     N";: NORMAL : GOSUB 10000
1100  GET W$: IF W$ =  CHR$ (27) THEN  GOSUB 8400
     : GOTO 870
1110  IF W$ < " " OR W$ > "^" OR X =  LEN (X$) +
     1 OR X = L + 1 THEN  PRINT BL$;: GOTO 1100
1120  IF  LEN (X$) = 1 THEN X$ = W$: GOSUB 8500:X
      = X + 1: GOSUB 10000: GOTO 1100
1130  IF X = 1 THEN X$ = W$ +  RIGHT$ (X$, LEN (X
     $) - 1): GOSUB 8500:X = X + 1: GOSUB 10000: GOTO
     1100
1140  IF X =  LEN (X$) THEN X$ =  LEFT$ (X$, LEN
     (X$) - 1) + W$: GOSUB 8500:X = X + 1: GOSUB
     10000: GOTO 1100
1150 X$ =  LEFT$ (X$,X - 1) + W$ +  RIGHT$ (X$, LEN
     (X$) - X): GOSUB 8500:X = X + 1: GOSUB 10000
     : GOTO 1100
```

```
1160  IF W$ <  >  CHR$ (18) THEN 1190
1170 X$ = T$(Y): GOSUB 8500: IF  LEN (X$) < X THEN
     X =  LEN (X$) + 1
1180  GOSUB 10000: GOTO 870
1190  IF W$ <  >  CHR$ (13) THEN 870
1200 T$(Y) = X$:T = Y:Y = NY:X = 1: GOSUB 10000: PRINT
     T$(NY);: IF  LEN (T$(NY)) < 40 THEN  CALL  -
     868
1210 Y = T: GOSUB 10000:X$ = "": GOTO 510
1550  IF FL THEN 510
1560  IF  LEN (X$) = L THEN 510
1570  PRINT W$;:X$ = X$ + W$: GOTO 510
2000 X = 1: FOR Y = 1 TO 23: GOSUB 10000: PRINT T
     $(Y);: IF  LEN (T$(Y)) < 40 THEN  CALL  - 86
     8
2010  NEXT : GOSUB 10000
2020  IF  LEN (T$(24)) < 40 THEN  PRINT T$(24);: CALL
      - 868:Y = T: GOSUB 10000: RETURN
2030  PRINT  LEFT$ (T$(24),39);: POKE 2039, ASC (
      RIGHT$ (T$(24),1)) + 128:Y = T: GOSUB 10000
     : PRINT BL$;: RETURN
2200  FOR I = 1 TO 24: IF  LEN (T$(I)) = 0 THEN 2
     230
2210  FOR J =  LEN (T$(I)) TO 1 STEP  - 1:W$ =  MID$
     (T$(I),J,1): IF W$ <  > " " THEN T$(I) =  LEFT$
     (T$(I),J): GOTO 2230
2220  NEXT J:T$(I) = ""
2230  NEXT I: FOR I = 24 TO 1 STEP  - 1: IF  LEN
     (T$(I)) <  > 0 THEN N = I: RETURN
2240  NEXT :N = 24:N = 0: RETURN
2300  VTAB 18: HTAB 3: INPUT "FILENAME:";FL$: IF
     FL$ = "" THEN  POP : GOTO 100
2310 W$ =  LEFT$ (FL$,1): IF W$ < "A" OR W$ > "Z"
      THEN 2300
2315  IF F1 THEN F1 = FALSE: RETURN
2320  PRINT : PRINT D$"OPEN "FL$: RETURN
3000  VTAB 19: HTAB 3: PRINT "ERASE FILE IN MEMOR
     Y ? "BL$;
3010  GET W$: IF W$ <  > "Y" THEN 100
3020  FOR I = 0 TO 25:T$(I) = "": NEXT :X = 1:Y =
     1: GOTO 100
3500  HOME : END
3600  HOME : VTAB 10: HTAB 3: INPUT "LEFT MARGIN
     ";LM
3610  HOME : VTAB 10: HTAB 5: PRINT "MAKE SURE TH
     E PRINTER IS READY": PRINT : PRINT : HTAB 5:
      PRINT "  TYPE <RETURN> WHEN READY": PRINT :
      HTAB 5: PRINT "     <ESC> FOR THE MENU "BL$;

3620  GET W$: IF W$ =  CHR$ (27) THEN 100
3625  IF W$ <  >  CHR$ (13) THEN 3620
3630  PRINT : PRINT D$"PR#1": GOSUB 2200: FOR I =
     1 TO N
```

```
3640  IF LM = 0 THEN 3660
3650  FOR J = 1 TO LM: PRINT " ";: NEXT
3660  PRINT T$(I): NEXT
3670  PRINT D$"PR#0": GOTO 100
4000  HOME : VTAB 8: INPUT "INITIAL LINE NUMBER:"
     ;LN$
4010  PRINT : INPUT "INCREMENT:";IN$
4020  GOSUB 2200
4030  FOR I = 1 TO N:
4999  STOP
5000  FOR I = 1 TO 20: IF  LEN (T$(I)) = 40 THEN
      PRINT T$(I);: GOTO 5020
5010  PRINT T$(I)
5020  NEXT
5030  GOTO 5030
6000  HOME : VTAB 7: HTAB 1: INPUT "INITIAL LINE
     NUMBER:";LN: IF LN <  = 10 OR LN > 63999 THEN
     6000
6005  PRINT : INPUT "INCREMENT:";IN
6006  PRINT : HTAB 16: INVERSE : PRINT "PROCESSIN
     G": NORMAL
6010 C = 1:IL = LN:PR$(1) =  STR$ (LN) + "HOME:?"
      + Q$:LN = LN + IN
6020  GOSUB 2200: IF N = 0 THEN  PRINT : INVERSE
     : PRINT "NOTHING TO CREATE "BL$;: NORMAL : GET
     W$: GOTO 100
6025  FOR I = 1 TO N: IF  LEN (T$(I)) = 0 THEN X$
      = LF$: GOSUB 7000: GOTO 6040
6030  FOR J = 1 TO  LEN (T$(I)): IF I <  > 24 OR
     J <  > 40 THEN 6037
6032 PR$(C) = PR$(C) + Q$ + ";": IF  LEN (PR$(C))
      > 225 THEN C = C + 1:PR$(C) =  STR$ (LN): GOTO
     6035
6033 PR$(C) = PR$(C) + ":"
6035 PR$(C) = PR$(C) + "POKE 2039," +  STR$ ( ASC
     ( RIGHT$ (T$(I),1)) + 128): GOTO 6060
6037 X$ =  MID$ (T$(I),J,1): GOSUB 7000: NEXT : IF
      LEN (T$(I)) < 40 AND I <  > N THEN X$ = LF$
     : GOSUB 7000
6040  NEXT : IF FL THEN PR$(C) = PR$(C) + Q$ + ";
     ": GOTO 6060
6060  PRINT : PRINT D$"OPEN DUMMY": PRINT D$"DELE
     TE DUMMY": PRINT D$"OPEN DUMMY": PRINT D$"WR
     ITE DUMMY"
6070  PRINT "NEW"
6071  PRINT "32000 J=1:DIM M%(2000):D$=CHR$(4):?D
     $"Q$"OPEN DUMMY1"Q$":?D$"Q$"READ DUMMY1"Q$":
     INPUT AC:FOR I=1 TO AC:INPUT M%(I):NEXT:?D$"
     Q$"CLOSE"Q$
6072  PRINT "32001 FOR I=2054 TO 10000:IF PEEK(I)
     =42 THEN POKE I,M%(J):AC=AC-1:J=J+1:GOTO 320
     03"
```

```
6074  PRINT "32003 NEXT"
6075  PRINT "32002 IF AC=0 THEN DEL 32000,32004"
6080 F2 = 1: IF F1 THEN PR$(0) = "10 Q$=CHR$(34)"
     :K = 0: GOTO 6087
6084 K = 1
6087  FOR I = K TO C: FOR J = 1 TO  LEN (PR$(I)):
     X$ =  MID$ (PR$(I),J,1): IF X$ <  > Q$ THEN
     6110
6090  PRINT X$;:F2 = F2 + 1: IF F2 = 3 THEN F2 =1
6100  GOTO 6140
6110  ON F2 GOTO 6120,6130
6120  PRINT X$;: GOTO 6140
6130  PRINT "*";:AC = AC + 1
6140  NEXT J: PRINT :F2 = 1: NEXT I
6145  PRINT "RUN 32000"
6150  PRINT D$"CLOSE"
6160 F2 = 1: PRINT D$"OPEN DUMMY1": PRINT D$"WRIT
     E DUMMY1": PRINT AC
6165  FOR I = 1 TO C: FOR J = 1 TO  LEN (PR$(I)):
     X$ =  MID$ (PR$(I),J,1): IF X$ <  > Q$ THEN
     6190
6170 F2 = F2 + 1: IF F2 = 3 THEN F2 = 1
6180  GOTO 6210
6190  ON F2 GOTO 6210,6200
6200  PRINT  ASC (X$)
6210  NEXT J:F2 = 1: NEXT : PRINT D$"CLOSE"
6220  PRINT : PRINT "  TO LOAD THE PROGRAM JUST C
     REATED,": PRINT : PRINT " LEAVE THE EDITOR A
     ND TYPE ";: INVERSE : PRINT "EXEC DUMMY";: NORMAL
     : GET W$
6230  GOTO 100
7000  IF X$ <  > Q$ THEN 7010
7003 F1 = TRUE: IF  LEN (PR$(C)) > 232 THEN  GOSUB
     7100:C = C + 1:PR$(C) =  STR$ (LN) + "?Q$" +
     Q$:LN = LN + IN: GOTO 7020
7005  IF  RIGHT$ (PR$(C),3) = "Q$" + Q$ THEN PR$(
     C) =  LEFT$ (PR$(C), LEN (PR$(C)) - 1) + "Q$
     " + Q$: GOTO 7020
7007 PR$(C) = PR$(C) + Q$ + "Q$" + Q$: GOTO 7020
7010 FL = TRUE:PR$(C) = PR$(C) + X$: IF  LEN (PR$
     (C)) >  = 235 THEN  GOSUB 7100:FL = FALSE:C =
     C + 1:PR$(C) =  STR$ (LN) + "?" + Q$:LN = LN
      + IN
7020  RETURN
7100  IF  RIGHT$ (PR$(C),1) = Q$ THEN PR$(C) =  LEFT$
     (PR$(C), LEN (PR$(C)) - 1): GOTO 7120
7110 PR$(C) = PR$(C) + Q$
7120 PR$(C) = PR$(C) + ";": RETURN
8400  HTAB 16: VTAB NY: INVERSE : PRINT " EDIT:ON
     ";: NORMAL : GOSUB 10000: RETURN
8500  HTAB 1: VTAB Y: IF Y < 24 OR  LEN (X$) < 40
      THEN 8505
```

```
8502  PRINT  LEFT$ (X$,39);: POKE 2039, ASC ( RIGHT$
     (X$,1)) + 128: GOTO 8510
8505  PRINT X$;: IF  LEN (X$) < 40 THEN  CALL  -
     868
8510  GOSUB 10000: RETURN
9000  IF Y = 24 THEN L = 39: RETURN
9010 L = 40: RETURN
9200  IF X = 1 THEN X$ =  RIGHT$ (X$, LEN (X$) -
     1): GOSUB 10000: PRINT X$" ";: GOSUB 10000: GOTO
     870
9500  IF X$ = "" THEN  RETURN
9510 T$(Y) = X$:X$ = "": IF  LEN (T$(Y)) = L THEN
      RETURN
9520  CALL  - 868: RETURN
10000  HTAB X: VTAB Y: RETURN
```

8

LOW RESOLUTION

When GR is executed, the screen is cleared to black, COLOR is set to 0 (black), and the mode is set to 40 by 40 dots with four lines of text; in order to keep the text in those four lines, a POKE 34, 20 is executed automatically, reducing the text window to four lines of text, at the bottom of the screen.

Every byte in the video memory is used to store two dots, using the two nibbles to store the color of each dot (see section 11.12). Since every nibble can store values from 0 to 15, 16 colors are available in this mode.

The most significant nibble corresponds to the top dot, and the least significant to the bottom one.

When the command TEXT is executed, the mode is changed back to text and the window is set to full screen, but no clearing takes place. Therefore, randomly placed characters are displayed in NORMAL, INVERSE, or FLASHING mode.

Table 6.1 shows the memory location of the switches that control the screen display.

8.1 SAVING LOW RESOLUTION SCREENS

Since the area where the low resolution pages reside is the same as for the text screen, BSAVE can be used to store in a file the contents of a screen of low resolution graphics.

If GR is in the mixed graphics and text mode, saving the entire memory range stores the text portions too. However, when the file is BLOADed back into memory, Applesoft doesn't remember the situation when the file was saved, and if the mode is not the same as when it was saved, what is shown on the screen will not correspond to the original image.

When you use BSAVE to store the low resolution screen, you must store the entire page, even if parts of the screen are without blocks, unless you study carefully the memory map and save the area of your interest. When you have one of these drawings that doesn't have many dots, you can save them in several ways: If you know the coordinates of the points, you can save the X and Y numbers in a regular text file; if you do not, you can scan the video memory and store only those bytes that are not 0 (which correspond to two black dots), when your background is black; if you are using another color, you can PEEK into one byte where you know the two dots are that background color (or you can calculate them). Then you can store the address and contents of those bytes that do not have that value.

Another way of saving the information in a low resolution page is to scan the entire screen with the function SCRN. If the color of the dot is not equal to the background, you store the position and color of the dot.

8.2 MOVING A LOW RESOLUTION IMAGE

There are several ways to move the entire low resolution screen. The easiest one is to use the regular text scroll, already available in machine language and executed with a CALL -912.

Since on entry to GR the text window is set to only four lines at the bottom, in order to have the entire window affected by the scroll, it must be reset to its original full-screen mode, with a POKE 34, 1. Having done this, a CALL -912 scrolls the screen two dots up, equivalent to one line of text.

The procedure described in program SCROLL (Section 7.3) can be readily used to scroll the low resolution screen, with the difference that the line that becomes empty does not correspond to blanks, but to INVERSE "@", a byte with value 0, the equivalent of two black dots, unless you are using a different background. If you are, you will have to calculate the value to POKE into this new empty line.

Since every character is equivalent to two dots stacked vertically, the vertical scrolls will move the screen two dots at a time.

The third way to do the scroll is to use the SCRN function. Following is an example of the program to scroll the screen vertically one dot column:

```
10 FOR I = 0 to 39 : FOR J = 0 to 47 : COLOR = SCRN I, J+1
20 PLOT I, J : NEXT : NEXT
30 COLOR = 0 : FOR I = 0 TO 39 : PLOT I, 48 : NEXT
```

If the background color is other than black, the color in line 30 should be set accordingly.

Doing the scroll in reverse order is trivial.

So far, all the graphics have been handled using dots. If your graphic will use many lines, a great deal of space can be saved when storing the screen, since the COLOR, the starting and ending points, and the displacement suffice to use the HLIN and VLIN commands. Speed is greatly increased, too.

8.3 LOW RESOLUTION EDITOR

As soon as the program is brought up, it sets low resolution with four lines of text, and the following menu is displayed:

1-EDIT
2-SAVE
3-LOAD
4-QUIT

1-EDIT

To the left of the text area DRAW is printed, indicating that at that moment any movement of the cursor will be PLOTted with the current COLOR, which is shown at the right hand side of the screen (the COLOR code, not the name).

To move the blinking cursor (and draw, if in the DRAW mode), the controls are the same as for the regular edition in Applesoft:

I - to go up
M - to go down
J - to go left
K - to go right

To change from the DRAW to the DON'T DRAW and vice versa, type D.

While in the DON'T DRAW mode, the cursor can move over already existing points, without altering them.

Pressing C, the COLOR can be changed (to a number between 0 and 15).

To return to the main menu, press <ESC>.

2-SAVE

The following four options are available:

1-BSAVE
2-AS DATA
3-CREATE PROGRAM
4-MENU

1-BSAVE. This option saves the entire video memory as a binary file. Before BLOADing it, set the low resolution mode, or else it will be displayed as text.

2-AS DATA. This option saves the number of points, their coordinates, and their COLOR.

3-CREATE PROGRAM. This option creates a file that, when EXECed, will become a program that draws the graphic you drew on the screen.

4-MENU. This option allows a return to the main menu.

3-LOAD

The following three options are available:

1-BLOAD
2-AS DATA
3-MENU

1-BLOAD. This option loads a binary file into the editor.
2-AS DATA. This option reads and draws a file created with the SAVE AS DATA option.
3-MENU. This option takes you back to the main menu.

4-QUIT

This option puts you back in TEXT, clears the screen, and exits the program.

Program Description

In line 5 some variables are initialized: TRUE and FALSE to their Applesoft equivalent. SC%( ) is DIMentioned to store the X and Y coordinates of the different points, and their COLOR.

The following menu with four options is PRINTed:

1-EDIT
2-SAVE
3-LOAD
4-QUIT

1-EDIT

Line 450 clears the text area. FL (the indicator of the DRAW, DON'T DRAW condition) makes subroutines 3010 or 3020 PRINT the appropriate messages. Subroutine 3000 PRINTs the current COLOR code, and the program enters the edit loop in line 500.

To be able to display a blinking cursor in low res, the program cannot stop to wait for a key to be pressed (the cursor will not blink while the program is waiting); therefore, subroutine 1000 is used instead of a GET or INPUT instruction.

In line 1000, if PEEK(−16384)>127, it means a key has been pressed, and the program branches to line 1050, where the keyboard strobe is reset, and W$ is set to the CHR$ typed.

If no key has been pressed, instruction 1010 increases K (the counter that keeps track of the blink). If it is less than five, it goes back to 1000. Otherwise, it changes the COLOR of the cursor from white to black, or vice versa (C1 is used as a flag to indicate the current COLOR of the cursor).

After RETURNing from subroutine 1000, the program compares W$ with I, J, K, and M, to see if it has to move; if it does, the movement is done only when in the proper margin (e.g., it will not move up if Y is in the top of the screen). After the movement or the bell that indicates the nearly-out-of-frame error, control is transferred back to line 500, and the cycle is repeated.

If C is pressed, line 590 calls subroutine 3000 and asks for a new COLOR code. It only accepts a number between 0 and 15, stores this value in the variable C, and sets the corresponding low resolution COLOR.

If <esc> is pressed, lines 630 to 645 PLOT or erase (PLOT with black) the block at the position where the cursor lies at that moment; this is to avoid the situation in which the cursor is in a different COLOR than the one indicated in the bottom lines of the text area. Control is transferred to the main menu.

If D is pressed, lines 655 and 660 set FL to the appropriate value, and the DRAW or DON'T DRAW sign is PRINTed.

2-SAVE

The following four options are available:

1-BSAVE
2-AS DATA
3-CREATE PROGRAM
4-MENU

1-BSAVE. Line 150 calls subroutine 3500, which prompts for a filename. If it gets an empty string, control is transferred to the main menu (notice the POP). If a filename is given, line 150 BSAVEs the video memory (starting at decimal 1024 and ending at 2039).

2-AS DATA. Line 180 calls subroutine 3500 (filename); if it is given one, it calls subroutine 4000, which scans the low resolution screen to find the coordinates and COLORs of the nonblack blocks. K is used as a counter for the array SC% ( ), where the X's, Y's, and COLORs will be stored. Using the variables I and G, every row and every column's COLOR is assigned to T with the SCRN function. If it is not 0 (black), the coordinates are stored in SC% (K, 0) and SC% (K, 1), and the COLOR in SC% (K, 2). Since this process can take a while and nothing seems to be happening, at the end of every row the speaker is toggled (V = PEEK (-16336)).

Lines 210 to 230 write N (the number of nonblack blocks), and SC% ( ) to the disk.

3-CREATE PROGRAM. This option creates a file that, when EXECed, will leave in memory a program that will draw the sketch currently on the screen. It asks for the filename and looks for the nonblack blocks (subroutines 3500 and 4000), prompts for an initial line number, and assigns the default 10 if it gets a <return>. Then it PRINTs the following instruction to the disk, assuming 10 has been chosen as the initial line number:

```
10 GR : READ N : FOR I = 1 TO N : READ C : COLOR = C : READ X :
   READ Y : PLOT X, Y : NEXT
```

It then sends N and every number in SC% ( ) (remember those are X, Y, and COLORs) to subroutine 5000, which converts the value sent to its corresponding STR$. If the length of BF$ (the temporary buffer where it has stored all the previous numbers, separated by commas) is more than an Applesoft line can be (or read by the EXEC command), it sends it to the disk, and adjusts BF$ to a new line number; otherwise, the new number is added to BF$. That way, consecutive line numbers of DATA are PRINTed to the file.

In line 350, when all SC% ( ) have been sent to line 5000, BF$ must be checked, to see if it is not empty. Not doing this would be equivalent to not CLOSEing a file: Some information might not have been PRINTed to the disk.

3-LOAD

There are three options:

1-BLOAD. Line 448 reads a binary file directly into memory.

2-AS DATA. A file is read into SC% ( ). Line 431 gives the option of drawing the image where it was when it was recorded (with the <RETURN> response), or to draw it in different coordinates. This might be useful when one image must be duplicated. With this option it can be drawn as many times as desired, in any position. Typing "M" makes line 432 go back to the main menu.

3-MENU.

```
1  REM  ** LOW RESOLUTION EDITOR **
5 TRUE = 1:FALSE = 0:D$ =  CHR$ (4): DIM SC%(1024
     ,2):Q$ =  CHR$ (34)
10  GR : COLOR= 15: HOME
20 BL$ =  CHR$ (7):C = 15
50  HOME : VTAB 21: HTAB 5: PRINT "1-EDIT": HTAB
     5: PRINT "2-SAVE"
53  HTAB 5: PRINT "3-LOAD": HTAB 5: PRINT "4-QUIT
     ";
55  GET W$: IF W$ < "1" OR W$ > "4" THEN  PRINT B
     L$;: GOTO 55
60  ON  VAL (W$) GOTO 450,100,400,800
100  HOME : VTAB 21: HTAB 5: PRINT "1-BSAVE": HTAB
     5: PRINT "2-AS DATA"
110  HTAB 5: PRINT "3-CREATE PROGRAM": HTAB 5: PRINT
     "4-MENU ";
120  GET W$:V =  VAL (W$): IF V < 1 OR V > 4 THEN
      PRINT BL$;: GOTO 120
130  ON V GOTO 150,180,300,50
150  GOSUB 3500: HOME : PRINT D$"BSAVE "FL$",A102
     4,L1024": GOTO 100
180  GOSUB 3500: GOSUB 4000
210  HOME : PRINT : PRINT D$"OPEN "FL$: PRINT D$"
     WRITE "FL$: PRINT N
220  FOR I = 1 TO N: PRINT SC%(I,0): PRINT SC%(I,
     1): PRINT SC%(I,2): NEXT : PRINT D$"CLOSE"
230  GOTO 50
300  GOSUB 3500: GOSUB 4000: HTAB 5: INPUT "INITI
     AL LINE NUMBER:";W$: IF W$ = "" THEN LN = 10
     : GOTO 310
305 LN =  VAL (W$)
310  PRINT D$"OPEN "FL$: PRINT D$"DELETE "FL$: PRINT
     D$"OPEN "FL$: PRINT D$"WRITE "FL$
320  PRINT  STR$ (LN)"GR:READ N:FOR I=1 TO N:READ
      C:COLOR=C:READ X:READ Y:PLOT X,Y:NEXT":LN =
     LN + 10
330 BF$ = "DATA" +  STR$ (N)
340  FOR I = 1 TO N: FOR J = 0 TO 2:AD = SC%(I,J)
     : GOSUB 5000
350  NEXT : NEXT : IF BF$ <  > "" THEN  PRINT  STR$
     (LN) + BF$
360  PRINT D$"CLOSE": GOTO 50
400  GOSUB 3500: HOME : VTAB 22: HTAB 5: PRINT "1
     -BLOAD": HTAB 5: PRINT "2-AS DATA": HTAB 5: PRINT
     "3-MENU";
410  GET W$:V =  VAL (W$): IF V < 1 OR V > 3 THEN
      PRINT BL$;: GOTO 410
420  ON V GOTO 448,425,100
425  HOME : PRINT : PRINT D$"OPEN "FL$: PRINT D$"
     READ "FL$: INPUT N: FOR I = 1 TO N
430  INPUT SC%(I,0): INPUT SC%(I,1): INPUT SC%(I,
     2): NEXT : PRINT D$"CLOSE"
```

```
431  HOME : VTAB 21: PRINT "DRAW AT (X,Y) <RETURN
     > FOR DEFAULT ": HTAB 15: PRINT "<M>       FO
     R MENU"
432  HTAB 5: INPUT "X:";X$: IF X$ = "M" THEN 50
434  HTAB 5: INPUT "Y:";Y$: IF X$ = "" AND Y$ = "
     " THEN DX = 0:DY = 0: GOTO 438
436 DX =  VAL (X$):DY =  VAL (Y$)
438  FOR I = 1 TO N:X = SC%(I,1) + DX:Y = SC%(I,2
     ) + DY: IF X < 0 OR X > 39 OR Y < 0 OR Y > 3
     9 THEN 442
440  COLOR= SC%(I,0): PLOT X,Y
442  NEXT : GOTO 431
445  FOR I = 1 TO N: COLOR= SC%(I,0): PLOT SC%(I,
     1),SC%(I,2): NEXT : GOTO 50
448  HOME : PRINT : PRINT D$"BLOAD "FL$: GOTO 50
450  HOME : ON FL + 1 GOSUB 3010,3020
470  GOSUB 3000: PRINT C;
500  GOSUB 1000: IF W$ <  > "I" THEN 520
505  IF Y = 0 THEN 2000
510  GOSUB 1200:Y = Y - 1: GOTO 1300
520  IF W$ <  > "M" THEN 540
525  IF Y = 39 THEN 2000
530  GOSUB 1200:Y = Y + 1: GOTO 1300
540  IF W$ <  > "J" THEN 560
545  IF X = 0 THEN 2000
550  GOSUB 1200:X = X - 1: GOTO 1300
560  IF W$ <  > "K" THEN 580
565  IF X = 39 THEN 2000
570  GOSUB 1200:X = X + 1: GOTO 1300
580  IF W$ <  > "C" THEN 620
590  GOSUB 3000
595  INPUT W$:C =  VAL (W$):FL = FALSE: GOSUB 301
     0
600  IF C < 0 OR C > 15 THEN 590
610  GOSUB 3000: PRINT C;: COLOR= C: GOTO 500
615  STOP
620  IF W$ <  >  CHR$ (27) THEN 650
630  IF FL THEN  COLOR= T: GOTO 645
640  COLOR= C
645  PLOT X,Y: GOTO 50
650  IF W$ <  > "D" THEN 670
655  IF FL THEN FL = FALSE: GOSUB 3010: GOTO 500
660 FL = TRUE: GOSUB 3020:T = 0: GOTO 500
670  IF W$ <  >  CHR$ (8) THEN 700
680 T = 0: GOTO 500
700  GOTO 500
800  TEXT : HOME : END
1000 W$ = "": IF  PEEK ( - 16384) >  = 128 THEN 1
     050
1010 K = K + 1: IF K < 5 THEN  RETURN
1020 K = 0: IF C1 THEN  COLOR= 0:C1 = 0: GOTO 1040
1030 C1 = TRUE: COLOR= 15
```

```
1040  PLOT X,Y: RETURN
1050 W$ =  CHR$ ( PEEK ( - 16384) - 128): POKE  -
     16368,0:K = K + 3: RETURN
1200  IF FL THEN  COLOR= T: PLOT X,Y: RETURN
1210  COLOR= C: PLOT X,Y: RETURN
1300  IF FL THEN T =  SCRN( X,Y)
1310  GOTO 500
1500  PLOT X,Y: GOTO 500
2000  PRINT BL$;: GOTO 500
3000  HTAB 25: VTAB 22: PRINT "COLOR : ";: CALL  -
     868: RETURN
3010  VTAB 22: HTAB 5: PRINT "DRAW       ";: RETURN

3020  VTAB 22: HTAB 5: PRINT "DON'T DRAW";: RETURN

3500  HOME : VTAB 22: HTAB 5: INPUT "FILE NAME:";
     FL$: IF FL$ = "" THEN  POP : GOTO 50
3510 W$ =  LEFT$ (FL$,1): IF W$ < "A" OR W$ > "Z"
      THEN  PRINT BL$;: GOTO 3500
3520  RETURN
4000 K = 1: FOR I = 0 TO 39: FOR J = 0 TO 39
4010 T =  SCRN( I,J): IF T <  > 0 THEN SC%(K,0) =
     T:SC%(K,1) = I:SC%(K,2) = J:K = K + 1
4020  NEXT :V =  PEEK ( - 16336): NEXT :N = K - 1
     : RETURN
5000 AD$ =  STR$ (AD): IF  LEN (BF$) +  LEN (AD$)
      < 230 THEN BF$ = BF$ + "," + AD$: RETURN
5010  PRINT  STR$ (LN) + BF$:LN = LN + 10:BF$ = "
     DATA " + AD$: RETURN
```

8.4 USING THE SECOND PAGE OF LOW RESOLUTION

Since the only difference between the text and low resolution modes is the situation of the switches described in Section 6.1, the description of the second page of text applies equally well to low resolution. See section 7.7.

8.5 MORE THAN TWO PAGES OF LOW RESOLUTION

Read sections 8.4 and 7.9.

HIGH RESOLUTION

9.1 PROTECTING THE HIGH RESOLUTION PAGES

If your program and the amount of memory it uses is not too big, the high resolution will behave normally. Unfortunately, it happens very often that you see how an array starts sharing the high resolution screen and both destroy each other. To see how it happens, run the following instruction:

```
10 HGR : DIM M (5000) : FOR I = 1 to 5000 step 30 : M (I) = I : NEXT
```

To protect your graphics (and your variables), you can set either LOMEM: or HIMEM: to addresses where you know nothing will happen. If you are using only the first page of memory, you can set LOMEM: to 16384; this will make Applesoft put all the variables in locations higher than this, and nothing will happen to the first page. If you are using the second page, you can either set HIMEM: to 16384, or LOMEM: to 24576, depending on the amount of memory in your system. Remember to set LOMEM: or HIMEM: at the very beginning of the program, or any variable with a previously assigned value will be lost.

9.2 HIGH RESOLUTION EDITOR

Instead of having to use huge numbers of HPLOTs to get a figure in your high resolution screen, an editor lets you draw any shape you want, in minutes. The basic principle of program HI RES EDITOR is the same used in low resolution: I, J, K, and M are used to move a blinking cursor, escape to go to a menu, and there is an option to change the color.

Line 10 sets BL$ to the bell character, DP to 1, and D$ to be used in the disk operations. Since the number of points in the screen is large, crossing the entire page one dot at a time can take a long time; that is why DP (the displacement increment) starts with 1, but can be changed to 10, to advance ten times as fast.

Line 20 sets page one of high resolution graphics and four lines of text, and prints the titles COLOR and DISPLACEMENT in VTAB 22.

Line 40 calls subroutine 4500 which prints the current values of PC (color) and DP (displacement). Line 100 is the beginning of the input loop. Subroutine 5000 reads the keyboard and makes the cursor blink by using C as a counter, and switching between colors 0 (black) and 3 (white) every time C becomes equal to 10.

Unlike the low resolution editor, when the input-and-blink subroutine RETURNs, it does not bring the character in a string variable. Since it entered as a number (the ASCII equivalent of the character typed), it is returned in the same way. All the comparisons are made using the ASCII codes of the characters. In practice, the difference is little (when it comes to speed and efficiency), but the program is harder to read, since a number is much less meaningful than a character. Yet, it is used in this program to show that it is equivalent.

When one character is received by this input subroutine, line 5010 resets the keyboard strobe, sets W to the value in the keyboard location (–16384), and sets HCOLOR to the value stored in (PC), since the current HCOLOR can be either black or white, because of the blinking cursor cycle.

If I is typed (ASCII 73), lines 110 and 120 decrement Y (the Y coordinate of the cursor) by DP (either 1 or 10), provided the resulting value is not outside the legal frame, and the dot at X, Y is HPLOTed with the current HCOLOR.

If J is typed (ASCII 74), lines 140 and 150 decrement X by DP, provided the result is still within the frame; X, Y is HPLOTed, and control is transferred back to line 100.

If K is typed (ASCII 75), lines 170 and 180 increment X when legal, HPLOT X, Y, and go back to line 100.

If M is typed (ASCII 77), lines 200 and 210 increment Y when legal, HPLOT X, Y, and go back to line 100.

If C is typed (ASCII 67), lines 230 and 240 toggle the HCOLOR (if it is black, it becomes white, and vice versa), call subroutine 4500 (PRINT in line 22 of the text part of the screen the new value of the HCOLOR), and jump back to line 100.

If D is typed, lines 260 and 270 toggle DP (the displacement increment) between its two possible values 1 and 10, call subroutine 4500 to PRINT the new value on the screen, and jump back to line 100.

If <esc> is typed (CHR$ (27)), line 310 jumps to line 500, the menu. The following options are displayed:

1-EDIT
2-SAVE
3-LOAD
4-CLEAR SCREEN
5-QUIT

1-EDIT

At the beginning of the program, in line 20, HGR was used to enter the first page of high resolution. This command has the side effect of clearing the page to black, which is good if you are starting. However, if you already have something in this page one, trying to reenter high resolution using HGR would erase the picture.

To go back to high resolution, a POKE −16304, 0 is executed. Then, line 520 jumps to line 30, in the editor, with the cursor blinking where it was before we pressed <esc> to go to the menu.

2-SAVE

Line 530 calls subroutine 6000, which prompts for a filename and jumps back to the menu if answered with a <return>. Then the entire high resolution screen is saved using the BSAVE command, with the address and length corresponding to page one.

3-LOAD

Line 540 calls subroutine 6000 for a filename and, if given one, it BLOADs the picture directly into page one.

4-CLEAR SCREEN

Lines 550 and 560 ask for confirmation (to avoid the paintul loss of a long drawing session because of a mistake) and if given, jump to line 20 where HGR erases the page and puts us back in the editor. Another way to do it is to execute a Call 62450.

All the other options available in the low resolution editor are not included here because there is no equivalent of the SCRN function in high resolution, and because those options are provided in the programs we will be discussing later.

```
1  REM  ** HIGH RESOLUTION EDITOR **
10  DIM M%(1000,1):BL$ =  CHR$ (7):DP = 1: POKE  -
     16297,0:D$ =  CHR$ (4)
20  HGR :PC = 3: HCOLOR= PC
30  HOME : HTAB 4: VTAB 22: INVERSE : PRINT "COLO
     R:";: HTAB 20: PRINT "DISPLACEMENT:";: NORMAL
40  GOSUB 4500
100  GOSUB 5000: IF W <  > 73 THEN 130
110  IF Y - DP < 0 THEN  PRINT BL$;: GOTO 100
120 Y = Y - DP: HPLOT X,Y: GOTO 100
130  IF W <  > 74 THEN 160
140  IF X - DP < 0 THEN  PRINT BL$;: GOTO 100
150 X = X - DP: HPLOT X,Y: GOTO 100
160  IF W <  > 75 THEN 190
170  IF X + DP > 279 THEN  PRINT BL$;: GOTO 100
180 X = X + DP: HPLOT X,Y: GOTO 100
190  IF W <  > 77 THEN 220
200  IF Y + DP > 159 THEN  PRINT BL$;: GOTO 100
210 Y = Y + DP: HPLOT X,Y: GOTO 100
220  IF W <  > 67 THEN 250
230  IF PC = 0 THEN PC = 3: HCOLOR= 3: GOSUB 4500
     : GOTO 100
240 PC = 0: HCOLOR= 0: GOSUB 4500: GOTO 100
250  IF W <  > 68 THEN 300
260  IF DP = 1 THEN DP = 10: GOSUB 4500: GOTO 100
270 DP = 1: GOSUB 4500: GOTO 100
300  IF W <  > 27 THEN 100
310  GOTO 500
500  HOME : POKE  - 16303,0: VTAB 8: PRINT "1-RET
     URN TO EDITOR": PRINT "2-SAVE": PRINT "3-LOA
     D": PRINT "4-CLEAR SCREEN": PRINT "5-QUIT";
510  GET W$:V =  VAL (W$): IF V < 1 OR V > 5 THEN
      PRINT BL$;: GOTO 510
515  ON V GOTO 520,530,540,550,3000
520  POKE  - 16304,0: GOTO 30
530  GOSUB 6000: PRINT : PRINT D$"BSAVE "FL$",A81
     92,L8192": GOTO 500
540  GOSUB 6000: PRINT : PRINT D$"BLOAD "FL$: GOTO
     500
550  PRINT : PRINT : PRINT "ERASE CURRENT PICTURE
      (Y/N) "BL$;
560  GET W$: IF W$ <  > "Y" THEN 500
570  GOTO 20
3000  TEXT : HOME : END
```

```
4500  VTAB 22: HTAB 11: PRINT PC;: HTAB 35: PRINT
     DP" ";: RETURN
5000 C = 0
5005  IF  PEEK ( - 16384) < 128 THEN 5020
5010  POKE  - 16368,0:W =  PEEK ( - 16384): HCOLOR=
     PC: HPLOT X,Y: RETURN
5020 C = C + 1: IF C < 10 THEN 5005
5030 C = 0: IF CO = 0 THEN CO = 1: GOTO 5050
5040 CO = 0
5050  IF CO = 0 THEN  HCOLOR= 3: GOTO 5070
5060  HCOLOR= 0
5070  HPLOT X,Y: GOTO 5000
6000  PRINT : PRINT : INPUT "  FILENAME : ";FL$: IF
     FL$ = "" THEN  POP : GOTO 500
6010  RETURN
```

9.3 TRANSFERRING PAGE 1 TO PAGE 2

Knowing where the high resolution pages start and end, it is trivial to transfer an image from one to the other. The easiest and fastest way is using BSAVE and BLOAD. The entire page with the picture is BSAVEd, and then BLOADed with the other page as its destination address. Another way is the one shown in the example. We assume the image is already in page one and will be transferred to page 2.

```
10 FOR I = 8192 TO 16383 : POKE I + 8192, PEEK (I) : NEXT
```

The transfer is achieved, but in a not very tidy way: There are 512 bytes in the page memory that were transferred and were not needed. Even though program TRANSFER is much longer than the single line instruction above, it not only does a neater job, but provides much information that will be very useful in coming sections.

Just as with the low resolution screen, the high resolution lines are not in sequential order in the memory. To see how they are distributed, run the following line:

```
10  HGR: FOR I = 8192 TO 16383 : POKE I, 127 : NEXT
```

Here, instead of having 48 lines, we have 192, which is a lot of data. It takes from lines 2000 to 2050, but since these values will be used in many other programs, typing them once pays off.

Line 5 sets LOMEM to 24576, since we are going to be using both high resolution pages. This way, the variables will be placed above the second page.

D is set to the distance between the two pages, so that every time this number is needed it doesn't have to be converted from a constant into its floating point equivalent. BL is used to click the speaker. Line 10 DIMensions M% ( ) to 191, to store the 192 of the leftmost byte in every line, and READs them.

Line 15 gives you the choice of using a picture already in memory, or loading it from disk (it must be stored in binary form).

Line 18 BLOADs the entered file into the first page, specifying the address ($2000, or decimal 8192); it does not matter if this image was BSAVEd from page 2.

For both options (picture in memory or BLOAD), HGR2 is set after the image is already in memory; this clears page 2, which is where page 1 will be transferred. Page 1 is not touched.

Line 25 uses I to keep track of the lines and J of the columns. M% (I) + J is the address, in memory, of the byte in line I and column J. The value found in every byte in the first page (PEEK (M% (I) + J)) is POKEd into its corresponding location in memory in page 2 by adding D (the distance between the two pages) to the memory address of the first page.

Line 26 makes the speaker click every time an entire line is transferred, to denote activity when a big area of the image is black and the program seems to be dead.

```
1  REM  ** TRANSFER **
5  LOMEM: 24576:D = 8192:BL =  - 16336
10  DIM M%(191):D$ =  CHR$ (4)
12  FOR I = 0 TO 191: READ M%(I): NEXT
15  TEXT : HOME : VTAB 8: PRINT "1-PICTURE IN MEM
     ORY": PRINT : PRINT "2-LOAD FROM DISK ";
16  GET W$: IF W$ = "1" THEN 20
17  IF W$ <  > "2" THEN 16
18  PRINT : PRINT : INPUT " FILENAME : ";FL$: IF
     FL$ = "" THEN  END
19  PRINT : PRINT D$"BLOAD "FL$",A$2000"
20  HGR2
25  FOR I = 0 TO 191: FOR J = 0 TO 39: POKE M%(I)
      + J + D, PEEK (M%(I) + J): NEXT
26 X =  PEEK (BL): NEXT
2000  DATA  8192,9216,10240,11264,12288,13312,143
     36,15360,8320,9344,10368,11392,12416,13440,1
     4464,15488,8448,9472,10496,11520,12544,13568
     ,14592,15616,8576,9600,10624,11648,12672,136
```

```
       96,14720,15744,8704,9728,10752,11776,12800,1
       3824
2010   DATA  14848,15872,8832,9856,10880,11904,129
       28,13952,14976,16000,8960,9984,11008,12032,1
       3056,14080,15104,16128,9088,10112,11136,1216
       0,13184,14208,15232,16256,8232,9256,10280,11
       304,12328,13352,14376,15400,8360,9384,10408,
       11432
2020   DATA  12456,13480,14504,15528,8488,9512,105
       36,11560,12584,13608,14632,15656,8616,9640,1
       0664,11688,12712,13736,14760,15784,8744,9768
       ,10792,11816,12840,13864,14888,15912,8872,98
       96,10920,11944,12968,13992,15016,16040,9000,
       10024
2030   DATA  11048,12072,13096,14120,15144,16168,9
       128,10152,11176,12200,13224,14248,15272,1629
       6,8272,9296,10320,11344,12368,13392,14416,15
       440,8400,9424,10448,11472,12496,13520,14544,
       15568,8528,9552,10576,11600,12624,13648,1467
       2,15696
2040   DATA  8656,9680,10704,11728,12752,13776,148
       00,15824,8784,9808,10832,11856,12880,13904,1
       4928,15952,8912,9936,10960,11984,13008,14032
       ,15056,16080,9040,10064,11088,12112,13136,14
       160,15184,16208,9168,10192,11216,12240,13264
       ,14288
2050   DATA  15312,16336
```

9.4 PAGES 3, 4, AND 5 OF HIGH RESOLUTION

Even though there is no provision to display more than two pages of high resolution, the activity switch can be set to draw in pages 3, 4, and 5. The location of these pages is found in Table 9.1.

Table 9.1. Location of high resolution pages.

PAGE	ADDRESS		ACTIVITY SWITCH
	DECIMAL	HEXADECIMAL	
1	8192	$2000	32
2	16384	$4000	64
3	24576	$6000	96
4	32768	$8000	128
5	40960	$10000	160

Since pages 4 and 5 are located where DOS resides, it is necessary to turn it OFF (see section 4.11). These pages can be used for production and temporary storage of graphics that are then transferred using a modification of program TRANSFER, or added to an existing page, using program COMBINE.

9.5 SECTION SCANNER

Using the information of program TRANSFER we can now store and use a section of any image. Since we are working with bytes and seven dots are stored in every byte, we will not be able to isolate an exact area of dots, but rather an area of bytes, which might be up to six columns bigger than the exact area wanted on every side, depending on where in the byte is the margin dot.

To determine which area is to be examined, a frame is drawn around the picture, and by pressing I, J, K, or M the corresponding side is moved to make the frame smaller. As with the high resolution editor, there is an option to move the line faster, pressing D. Once the frame is where desired, <esc> starts the scanning process. Since some of the lines of the frame may be located in the same byte of the points adjacent to it and we do not want the frame to be in the final portion of the picture, before starting the process we make a copy of page 1 into page 2 using an exact copy of program TRANSFER, and draw the frame in page 2. That way, the limits can be found without altering page 1 in any way.

Line 5 sets LOMEM:, D and BL as program TRANSFER. Line 10, in addition to DIMensioning M%( ) for the memory locations, DIMensions L% ( ) and P% ( ); P% ( ) will store the address of every nonzero byte found (when a byte contains a zero it is black and does not affect the graphic). L% ( ) will store its value, which when POKEd into any byte of the high resolution area will redraw a seven dot area.

Lines 12 through 26 are not different from those of program TRANSFER; line 26, though, adds a new feature: It sets Applesoft to send everything HPLOTted to page 2.

Line 30 draws the original frame, and line 70 starts the frame-alteration loop.

If W$ = "I," line 80 increments Y1 (the top margin of the frame) by F, which can be either 1 or 10, and draws the new line back to line 70.

Lines 90 through 110 do the same as 80 with the other three margins of the frame, in their respective directions. Notice that no checking is being made as to legal values. We assume that this situation will never occur, since to have margins beyond the legal area implies that there is no longer a frame, and in this program it does not make sense.

If W$ = D, lines 130 and 140 toggle F between 1 and 10, to increase or decrease the speed of the margin movement.

Finally, if <esc> is typed, line 1030 starts the actual scanning. The I-loop (rows) must go from Y1 to Y2, but the J-loop limits, having X1 and X2 reflecting the actual coordinates of the screen (0 – 280), must be converted to their byte equivalent. Since there are seven dots per byte, the INT( ) is the distance in bytes from the left margin to the one containing X1. The byte for X2 is calculated the same way. Line 1040 stores in W the PEEK of the corresponding byte. If its value is not zero, its address is stored in P% (C) and its read value in L% (C). C is incremented.

Line 1050 ends the loops emitting a click every time a line is completed, and sets N to the number of nonzero bytes found, when all the area inside the frame has been analyzed.

Line 1060 sets the activity switch to page 1, and calls the subroutine that clears the page.

Line 1070 goes into TEXT mode to indicate that it has finished the scanning, and displays the following six options:

1-ORIGINAL POSITION
2-CHANGE X
3-CLEAR SCREEN
4-VIEW
5-SAVE
6-QUIT

1-ORIGINAL POSITION

Line 1100 sets X to 0 and jumps to line 1170, which calls subroutine 1220 to display temporarily a sign to indicate that by pressing <esc> the program returns to the menu. Line 1230 changes the mode into graphics. The loop in lines 1180 and 1190 uses the values of P% ( ) as addresses to POKE the values stored in L% ( ). Since in this case X is 0, the drawing corresponds to its original position in the screen.

2-CHANGE X

Line 1110 asks for the new X and jumps to line 1170. The process is identical to the one in choice 1, adding the value entered in X to the address where the values are going to be POKEd. Therefore, a negative X will shift the new picture X bytes to the left (or X ✱ 7), and a positive X will shift it proportionally to the right.

3-CLEAR SCREEN

Line 1120 asks for confirmation and, if given, line 1130 jumps to line 1060 which erases page 1.

4-VIEW

Line 1140 jumps to subroutine 1220 to display the <esc> sign and go into graphics. Then it goes to line 1200 and waits for an <esc> to return to the menu.

5-SAVE

Line 1150 asks for a filename and returns to the menu if answered with a <return>. Otherwise, it BSAVEs page 1 under the given name.

When using the second option, Change X, there are limitations due to the fact that the representation of the page is not sequential in memory. If X is small enough to make the drawing still fit in the same line of 40 bytes, every part of the picture will be contained in one single area. When the new addresses correspond to a different line, different things can happen, depending on the position in memory: The new graph can be perfect, it can be split into two or more parts in different parts of the screen, or part of it can be drawn outside the page.

```
1   REM  ** SCANNER **
5   LOMEM: 24576:D = 8192:BL =  - 16336
10  X1 = 0:X2 = 279:Y1 = 0:Y2 = 191:F = 1: DIM M%(
      191),L%(3000),P%(3000):D$ =  CHR$ (4): FOR I
       = 0 TO 191: READ M%(I): NEXT
15   TEXT : HOME : VTAB 8: PRINT "1-PICTURE IN MEM
      ORY": PRINT : PRINT "2-LOAD FROM DISK ";
16   GET W$: IF W$ = "1" THEN 20
17   IF W$ <  > "2" THEN 16
```

```
18  PRINT : PRINT : INPUT "  FILENAME : ";FL$: IF
     FL$ = "" THEN  END
19  PRINT : PRINT D$"BLOAD "FL$",A$2000"
20  HGR2
25  FOR I = 0 TO 191: FOR J = 0 TO 39: POKE M%(I)
      + J + D, PEEK (M%(I) + J): NEXT
26 X =  PEEK (BL): NEXT
28  POKE 230,64
30  HCOLOR= 3: HPLOT 0,0 TO 279,0 TO 279,191 TO 0
     ,191 TO 0,0
70  GET W$
80  IF W$ = "I" THEN Y1 = Y1 + F: HPLOT X1,Y1 TO
     X2,Y1: GOTO 70
90  IF W$ = "M" THEN Y2 = Y2 - F: HPLOT X1,Y2 TO
     X2,Y2: GOTO 70
100  IF W$ = "J" THEN X1 = X1 + F: HPLOT X1,Y1 TO
     X1,Y2: GOTO 70
110  IF W$ = "K" THEN X2 = X2 - F: HPLOT X2,Y1 TO
     X2,Y2: GOTO 70
120  IF W$ <  > "D" THEN 150
130  IF F = 1 THEN F = 10: GOTO 70
140 F = 1: GOTO 70
150  IF W$ <  >  CHR$ (27) THEN 70
1030  FOR I = Y1 TO Y2: FOR J =  INT (X1 / 7) TO
      INT (X2 / 7)
1040 Q = M%(I) + J:W =  PEEK (Q): IF W > 0 THEN P
     %(C) = Q:L%(C) = W:C = C + 1
1050  NEXT :X =  PEEK (BL) +  PEEK (BL): NEXT :N =
     C
1060  POKE 230,32: CALL 62450
1070  TEXT : HOME : VTAB 6: PRINT "1-ORIGINAL POS
     ITION": PRINT : PRINT "2-CHANGE X": PRINT : PRINT
     "3-CLEAR SCREEN": PRINT : PRINT "4-VIEW": PRINT
     : PRINT "5-SAVE": PRINT : PRINT "6-QUIT ";
1080  GET W$: IF W$ < "1" OR W$ > "6" THEN 1080
1090  ON  VAL (W$) GOTO 1100,1110,1120,1140,1150,
     1240
1100 X = 0: GOTO 1170
1110  PRINT : PRINT : INPUT "  NEW X=";X: GOTO 11
     70
1120  PRINT : PRINT : PRINT "  ERASE PICTURE IN M
     EMORY ? ";: GET W$: IF W$ <  > "Y" THEN 1070

1130  GOTO 1060
1140  GOSUB 1220: GOTO 1200
1150  PRINT : PRINT : INPUT "  FILENAME : ";FL$: IF
     FL$ = "" THEN 1070
1160  PRINT : PRINT D$"BSAVE "FL$",A$2000,L$2000"
     : GOTO 1070
1170  GOSUB 1220
1180  FOR I = 0 TO N
1190  POKE P%(I) + X,L%(I): NEXT
```

```
1200  GET W$: IF W$ <   >  CHR$ (27) THEN 1200
1210  GOTO 1070
1220  PRINT : PRINT : INVERSE : PRINT "TYPE <ESC>
      FOR THE MENU";: NORMAL : FOR I = 1 TO 2000:
      NEXT
1230  POKE  - 16304,0: RETURN
1240  HOME : END
2000  DATA  8192,9216,10240,11264,12288,13312,143
     36,15360,8320,9344,10368,11392,12416,13440,1
     4464,15488,8448,9472,10496,11520,12544,13568
     ,14592,15616,8576,9600,10624,11648,12672,136
     96,14720,15744,8704,9728,10752,11776,12800,1
     3824
2010  DATA  14848,15872,8832,9856,10880,11904,129
     28,13952,14976,16000,8960,9984,11008,12032,1
     3056,14080,15104,16128,9088,10112,11136,1216
     0,13184,14208,15232,16256,8232,9256,10280,11
     304,12328,13352,14376,15400,8360,9384,10408,
     11432
2020  DATA  12456,13480,14504,15528,8488,9512,105
     36,11560,12584,13608,14632,15656,8616,9640,1
     0664,11688,12712,13736,14760,15784,8744,9768
     ,10792,11816,12840,13864,14888,15912,8872,98
     96,10920,11944,12968,13992,15016,16040,9000,
     10024
2030  DATA  11048,12072,13096,14120,15144,16168,9
     128,10152,11176,12200,13224,14248,15272,1629
     6,8272,9296,10320,11344,12368,13392,14416,15
     440,8400,9424,10448,11472,12496,13520,14544,
     15568,8528,9552,10576,11600,12624,13648,1467
     2,15696
2040  DATA  8656,9680,10704,11728,12752,13776,148
     00,15824,8784,9808,10832,11856,12880,13904,1
     4928,15952,8912,9936,10960,11984,13008,14032
     ,15056,16080,9040,10064,11088,12112,13136,14
     160,15184,16208,9168,10192,11216,12240,13264
     ,14288
2050  DATA  15312,16336
```

9.6 DOT ORGANIZATION

To understand clearly the next two programs, it is important to have a clear idea of how the dots of the high resolution pictures are stored in memory. If you are not familiar with binary numbers, read Section 11.12.

Each of the 192 lines of high resolution consists of 40 bytes, each one capable of displaying seven dots. The least significant bit (the rightmost one) of the first byte corresponds to the first dot of the line, the second to the second, and so on until the seventh. Then, the least significant bit of the next byte corresponds to the eighth dot, the second to the ninth dot, and so on; the eighth bit of every byte is used to determine the color of the dot.

For more detailed information on the colors in high resolution, read page 19 of the Apple Reference Manual.

The most important thing to know for the next two sections is that a bit with value 0 is always black. These programs will act on dots that are turned ON or OFF without concern for the color.

9.7 BIT SCANNER

To find out if a particular dot is ON in the screen, the general procedure is to calculate the exact byte where that dot is in memory, to PEEK into that address, to convert the number found into its binary equivalent and, last, to see if the bit corresponding to the dot is 1 or 0.

Program BIT SCANNER assumes the picture is already in page 1; it may be created with the editor, BLOADed from a disk, or created by another program.

Lines 5 and 10 do the same initializations of the two previous programs. Lines 20 through 23 set page one of high resolution without text, without erasing the screen. Line 25 sets the activity switch to page 1. Line 30 draws the frame, and lines 80 through 150 let you adjust it to the desired size.

When the frame is ready and <esc> is pressed, line 450 goes into TEXT and PRINTs the total number of lines (the right margin minus the left minus 2), and the title "ANALYZING LINE #", to show you how far it is in the process and how much is still to go.

Line 500 starts the Y-loop (the lines), prints the number of the line being analyzed, and starts the X-loop (the columns). Having X and Y, line 510 calls subroutine 10000, whose task is to inform if the dot at X, Y is ON or OFF.

In low resolution, the function SCRN returns the color of a particular point. Unfortunately, there is not an equivalent for high resolution; in this program, it is done by subroutine 10000. On return from it, the boolean variable HSCR will be either TRUE or FALSE (TRUE when the dot is ON).

Line 10000 sets BYTE to the distance between the left margin of that line (whose address we have in M% ( )) and the byte where the point X is. BIT is set to the absolute position of X in the byte. Line 10010 compares BYTE with the byte previously analyzed. Converting a number into its binary equivalent is a slow process, and if the byte under consideration has already been converted, the operation need not be repeated.

Line 10015 PEEKs into the absolute address in memory where dot X, Y is (the address of the left margin of the line in M% (Y) plus the distance in BYTE), and stores that value in N. A$ is set to " ". It will keep the 0's and 1's of the binary equivalent of N, and we want it to be free of any garbage at the start.

Lines 10020 through 10060 convert it into its string binary equivalent using the technique described in section 11.13, storing in A$ the 0's and 1's in reversed order (the bits correspond to the dots in reverse order). Line 10065 sets PBYTE (the previous byte) to BYTE, and line 10070 compares the character in position BIT of A$ with 1 and 0, and sets HSCR to TRUE or FALSE accordingly.

Back in line 510, if HSCR is TRUE (the dot is ON), its coordinates are stored in X% ( ) and Y% ( ) and C is incremented; if the dot is OFF, nothing is stored.

At the end of the two loops, line 530 sets N to the number of ON bits encountered.

Line 540 prints a menu with the following six options:

1-VIEW
2-PURGE
3-RELOCATE
4-SAVE
5-LOAD
6-QUIT

1-VIEW

Line 570 asks for a color; this allows you to see a figure in any color, regardless of how it was originally displayed on the screen. Subroutine 800 informs, for a short period, that <esc> returns the program to the menu. Line 580 calls subroutine 750 with X and Y equal to 0.

Line 750 sets full page high resolution graphics and HCOLOR to the one chosen. Then it HPLOTs all the N points, using the coordinates in X% ( ) and Y% ( ). Therefore, the portion of the picture originally inside the frame is redrawn. X and Y are added to every X% and Y% to change the position of the picture. In this case, since both are 0, the addition has no effect. Line 760 waits until <esc> is pressed.

2-PURGE

To be able to draw this new picture in different areas of the screen, it is inconvenient to have to add negative numbers to move it to the left and positive ones to the right, not knowing which will exceed the limits of the screen.

Therefore, it is very useful to have the picture at the extreme left-top corner, where the smallest X% ( ) and Y% ( ) are 0.

To do this, line 610 assumes that X% (0) and Y% (0) are the smallest of each one of its arrays, and proceeds to compare those values with the rest. Whenever a smaller one is found, it replaces the previous one, so that at the end of the loop, in line 630, X and Y have the smallest numbers of each X% ( ) and Y% ( ). Then, another loop subtracts this number from every number in the arrays, so that the entire picture is shifted to 0, 0.

3-RELOCATE

To draw the picture in a new location, line 640 asks for a new X and Y. If the arrays have been already purged, X and Y will be added to the top and left margins of the picture; otherwise, the result depends on the original values. Subroutine 750 draws the picture in this new location (this time adding X and Y to X% ( ) and Y% ( )), and line 650 goes back to the menu.

4-SAVE

Line 660 calls subroutine 780 to get the filename, which if empty returns the program to the menu; otherwise, the file is OPENed and N is PRINTed. Every element in X% ( ) and Y% ( ) is PRINTed too, and the file is CLOSEd.

5-LOAD

Line 680 calls subroutine 780 for a filename, OPENs the file, reads N and all the values of X% ( ) and Y% ( ), CLOSEs the file, and goes back to the menu.

In the high resolution editor the only way to SAVE the picture is using BSAVE, because we do not have the equivalent of the SCRN function of low resolution. Now, we can save the exact points, to use them later to create a DATA table, to read them directly from the disk and HPLOT them, etc.

Subroutine 10000 (which will be called HSCR from now on) gives us an enormous range of new possibilities. Here are some:

a) Using a printer without graphic capabilities, all or part of a high resolution screen can be scanned line by line and, if a dot is found to be ON, a certain character is PRINTed. Otherwise, a blank simulates an OFF dot. The distance between lines must be taken into consideration, because it can affect the scale of the X and Y axis; if this distance can be modified, make it correspond to the length of a character.

b) If every dot that is found to be ON is not drawn with one but four dots on the screen and the distances are multiplied by two (so as to simulate four-dotted black dots), the picture will be scaled up to twice its original size. Of course, any number can be used as a scaling factor.

c) Transferring graphics between low and high resolution and vice versa is a straightforward matter, since the X% ( ) and Y% ( ) values are absolute coordinates in an X, Y plane.

```
1  REM  ** BIT SCANNER **
5  LOMEM: 16385:D$ =  CHR$ (4)
10 X1 = 0:X2 = 279:Y1 = 0:Y2 = 191:F = 1: DIM M
    %(191),X%(2000),Y%(2000)
15  FOR I = 0 TO 191: READ M%(I): NEXT
20  POKE  - 16304,0
21  POKE  - 16302,0
22  POKE  - 16300,0
23  POKE  - 16297,0
25  POKE 230,32
30  HCOLOR= 3: HPLOT 0,0 TO 279,0 TO 279,191 TO
    0,191 TO 0,0
70  GET W$
80  IF W$ = "I" THEN Y1 = Y1 + F: HPLOT X1,Y1 TO
    X2,Y1: GOTO 70
90  IF W$ = "M" THEN Y2 = Y2 - F: HPLOT X1,Y2 TO
    X2,Y2: GOTO 70
100  IF W$ = "J" THEN X1 = X1 + F: HPLOT X1,Y1 TO
    X1,Y2: GOTO 70
```

```
110  IF W$ = "K" THEN X2 = X2 - F: HPLOT X2,Y1 TO
     X2,Y2: GOTO 70
120  IF W$ <  > "D" THEN 150
130  IF F = 1 THEN F = 10: GOTO 70
140 F = 1: GOTO 70
150  IF W$ <  >  CHR$ (27) THEN 70
450  TEXT : HOME : VTAB 10: HTAB 5: PRINT "TOTA
     L NUMBER OF LINES:"Y2 - Y1 - 1
460  HTAB 5: VTAB 12: PRINT "ANALIZING LINE # "

500  FOR Y = Y1 + 1 TO Y2 - 1: HTAB 22: VTAB 12
     : PRINT Y - Y1: FOR X = X1 + 1 TO X2 - 1
510  GOSUB 10000: IF HSCR THEN X%(C) = X:Y%(C) =
     Y:C = C + 1
520  NEXT : NEXT
530 N = C - 1
540  TEXT : HOME : VTAB 6: PRINT "1-VIEW": PRINT
     : PRINT "2-PURGE": PRINT : PRINT "3-RELOCA
     TE": PRINT : PRINT "4-SAVE": PRINT : PRINT
     "5-LOAD": PRINT : PRINT "6-QUIT ";
550  GET W$: IF W$ < "1" OR W$ > "6" THEN 550
560  ON  VAL (W$) GOTO 570,610,640,660,680,700
570  PRINT : PRINT : INPUT "COLOR : ";C: GOSUB
     800
580 X = 0:Y = 0: GOSUB 750: GOTO 540
610  PRINT : PRINT : INVERSE : PRINT "PURGING";
     : NORMAL :X = X%(0):Y = Y%(0): FOR I = 1 TO
     N: IF X%(I) < X THEN X = X%(I)
620  IF Y%(I) < Y THEN Y = Y%(I)
630  NEXT : FOR I = 0 TO N:X%(I) = X%(I) - X:Y%
     (I) = Y%(I) - Y: NEXT : GOTO 540
640  PRINT : PRINT : INPUT "NEW X=";X:: PRINT :
      INPUT "NEW Y=";Y: GOSUB 750
650  GOTO 540
660  GOSUB 780: PRINT : PRINT D$"OPEN "FL$: PRINT
     D$"WRITE "FL$: PRINT N
670  FOR I = 0 TO N: PRINT X%(I): PRINT Y%(I): NEXT
     : PRINT D$"CLOSE": GOTO 540
680  GOSUB 780: PRINT : PRINT D$"OPEN "FL$: PRINT
     D$"READ "FL$: INPUT N
690  FOR I = 0 TO N: INPUT X%(I): INPUT Y%(I): NEXT
     : PRINT D$"CLOSE": GOTO 540
700  HOME : END
730  HGR : HCOLOR= 3: FOR I = 0 TO N: HPLOT X%(I)
     ,Y%(I): NEXT
750  HGR : POKE  - 16302,0: HCOLOR= C: FOR I = 0 TO
     N: HPLOT X%(I) + X,Y%(I) + Y: NEXT
760  GET W$: IF W$ <  >  CHR$ (27) THEN 760
770  RETURN
780  PRINT : PRINT : INPUT "  FILENAME : ";FL$: IF
     FL$ = "" THEN  POP : GOTO 540
```

```
790 W$ =  LEFT$ (FL$,1): IF W$ < "A" OR W$ > "Z" THEN
     780
795  RETURN
800  PRINT : PRINT : INVERSE : PRINT "TYPE <ESC>
     FOR THE MENU";: NORMAL : FOR I = 1 TO 1500: NEXT
     : RETURN
10000 BYTE =  INT (X / 7):BIT = (X - BYTE * 7) +
     1
10010  IF PBYTE = BYTE THEN 10070
10015 N =  PEEK (M%(Y) + BYTE)
10017 A$ = ""
10020  FOR I = 7 TO 0 STEP  - 1
10030 DI =  INT (N / 2 ^ I)
10040 A$ =  STR$ (DI) + A$
10050 N = N -  INT (DI * 2 ^ I)
10060  NEXT
10065 PBYTE = BYTE
10070  IF  MID$ (A$,BIT,1) = "1" THEN HSCR = 1: RETURN

10080 HSCR = 0: RETURN
20000  DATA  8192,9216,10240,11264,12288,13312,14
     336,15360,8320,9344,10368,11392,12416,13440,
     14464,15488,8448,9472,10496,11520,12544,1356
     8,14592,15616,8576,9600,10624,11648,12672,13
     696,14720,15744,8704,9728,10752,11776,12800,
     13824
20010  DATA  14848,15872,8832,9856,10880,11904,12
     928,13952,14976,16000,8960,9984,11008,12032,
     13056,14080,15104,16128,9088,10112,11136,121
     60,13184,14208,15232,16256,8232,9256,10280,1
     1304,12328,13352,14376,15400,8360,9384,10408
     ,11432
20020  DATA  12456,13480,14504,15528,8488,9512,10
     536,11560,12584,13608,14632,15656,8616,9640,
     10664,11688,12712,13736,14760,15784,8744,976
     8,10792,11816,12840,13864,14888,15912,8872,9
     896,10920,11944,12968,13992,15016,16040,9000
     ,10024
20030  DATA  11048,12072,13096,14120,15144,16168,
     9128,10152,11176,12200,13224,14248,15272,162
     96,8272,9296,10320,11344,12368,13392,14416,1
     5440,8400,9424,10448,11472,12496,13520,14544
     ,15568,8528,9552,10576,11600,12624,13648,146
     72,15696
20040  DATA  8656,9680,10704,11728,12752,13776,14
     800,15824,8784,9808,10832,11856,12880,13904,
     14928,15952,8912,9936,10960,11984,13008,1403
     2,15056,16080,9040,10064,11088,12112,13136,1
     4160,15184,16208,9168,10192,11216,12240,1326
     4,14288
20050  DATA  15312,16336
```

9.8 NEGATIVE HI-RES

Program BIT SCANNER lets you change the color in a particular picture, but it does not allow you to invert them, so that all the dots that are turned ON are turned OFF and the colors are complemented.

It is not too difficult to see how to do it. Using subroutine HSCR, A$ could be changed so that every 1 becomes a 0 and vice versa, and then every 1 could be multiplied by its corresponding power of two to get the value of the whole byte, which would then be POKEd back into its original position.

It so happens that there are only 256 different values in a byte, and therefore only 256 different combinations of 0's and 1's. If we take every one of those combinations and calculate the value of the binary number in which every bit has been complemented, we find the following relation:

VALUE	COMPLEMENT
0	255
1	254
2	253
.	.
.	.
254	1
255	0

Therefore, we can PEEK into every memory location of the page much faster than using subroutine HSCR, and POKE the complement of the number we find there.

In program NEGATIVE, N% ( ) is used to store the complement of every number from 0 to 255, so that when a 1 is found in a specific byte, for example, N% (1) contains the complement of 1, and the calculations must be done only once.

Line 10 PRINTs a title to show that the program is not "dead," while the usual M% ( ) is READ and the calculations are performed to put the complements of the indices into N% ( ), which is done by subtracting the number from 255, in lines 20 and 30.

Line 50 gives the option of BLOADing a picture from disk or continuing if the image is already in the high resolution area. Line 100 calls subroutine 260, which enters page one of high resolution with no text, without erasing the picture.

Line 110 starts the I-loop (lines) and J-loop (columns). Line 120 PEEKs into the memory location corresponding to each byte (there is no bit-level operation needed here, which is why J only goes from 0 to 39), and the value found is used as the index for N% ( ), which is POKEd back into the same location.

Lines 190 and 160 display a menu with the following three options:

1-DISPLAY PICTURE
2-SAVE
3-QUIT

1-DISPLAY PICTURE

Line 190 shows temporarily the reminder that <esc> makes the program go back to the menu, calls subroutine 260 which turns on the high resolution, and waits in line 210 until an <esc> is pressed.

2-SAVE

Line 230 asks for a filename and returns to the menu if answered with an empty string. Line 240 BSAVEs the high resolution page.

If only a part of the screen is to be changed into its negative, a technique similar to the one used in program SCANNER can help, since it worked at the byte level. If bit accuracy is needed, a combination of programs SCANNER and BIT SCANNER could be used: Those bytes whose seven bits have to be complemented can be treated as in program NEGATIVE, and the isolated bits can be examined and changed using HSCR.

```
1  REM  ** NEGATIVE **
10  LOMEM: 16384: HOME : INVERSE : HTAB 13: PRINT
     "HI-RES NEGATIVE": NORMAL
20 D$ =  CHR$ (4): DIM M%(191),N%(255): FOR I = 0
      TO 255
30 N%(I) = 255 - I: NEXT
40  FOR I = 0 TO 191: READ M%(I): NEXT
50  VTAB 10: HTAB 3: PRINT "1-PICTURE ALREADY IN
     MEMORY": VTAB 12: HTAB 3: PRINT "2-LOAD PICT
     URE FROM DISK ";
60  GET W$:V =  VAL (W$): IF V < 1 OR V > 2 THEN
     60
```

```
70   ON V GOTO 100,80
80   VTAB 14: HTAB 5: INPUT "FILENAME:";W$
90   PRINT : PRINT  CHR$ (4);"BLOAD "W$",A$2000"
100   GOSUB 260
110   FOR I = 0 TO 191: FOR J = 0 TO 39
120 W = M%(I) + J: POKE W,N%( PEEK (W))
130   NEXT : NEXT
140   HOME : TEXT
150   HTAB 3: VTAB 10: PRINT "1-DISPLAY PICTURE": HTAB
      3: VTAB 12: PRINT "2-SAVE TO DISK ";
160   HTAB 3: VTAB 14: PRINT "3-QUIT ";
170   GET W$:V =  VAL (W$): IF V < 1 OR V > 3 THEN
      170
180   ON V GOTO 190,230,250
190   HTAB 5: VTAB 16: INVERSE : PRINT "PRESS <ESC
      > FOR MENU": NORMAL
200   FOR I = 1 TO 2000: NEXT : GOSUB 260
210   GET W$: IF W$ <  >  CHR$ (27) THEN 210
220   GOTO 140
230   VTAB 16: HTAB 5: INPUT "FILENAME:";W$
240   PRINT : PRINT D$"BSAVE "W$",A$2000,L$2000": GOTO
      140
250   HOME : END
260   POKE  - 16304,0
310   POKE  - 16302,0
320   POKE  - 16300,0
330   POKE  - 16297,0
340   RETURN
20000  DATA  8192,9216,10240,11264,12288,13312,14
      336,15360,8320,9344,10368,11392,12416,13440,
      14464,15488,8448,9472,10496,11520,12544,1356
      8,14592,15616,8576,9600,10624,11648,12672,13
      696,14720,15744,8704,9728,10752,11776,12800,
      13824
20010  DATA  14848,15872,8832,9856,10880,11904,12
      928,13952,14976,16000,8960,9984,11008,12032,
      13056,14080,15104,16128,9088,10112,11136,121
      60,13184,14208,15232,16256,8232,9256,10280,1
      1304,12328,13352,14376,15400,8360,9384,10408
      ,11432
20020  DATA  12456,13480,14504,15528,8488,9512,10
      536,11560,12584,13608,14632,15656,8616,9640,
      10664,11688,12712,13736,14760,15784,8744,976
      8,10792,11816,12840,13864,14888,15912,8872,9
      896,10920,11944,12968,13992,15016,16040,9000
      ,10024
20030  DATA  11048,12072,13096,14120,15144,16168,
      9128,10152,11176,12200,13224,14248,15272,162
      96,8272,9296,10320,11344,12368,13392,14416,1
      5440,8400,9424,10448,11472,12496,13520,14544
      ,15568,8528,9552,10576,11600,12624,13648,146
      72,15696
```

```
20040  DATA  8656,9680,10704,11728,12752,13776,14
       800,15824,8784,9808,10832,11856,12880,13904,
       14928,15952,8912,9936,10960,11984,13008,1403
       2,15056,16080,9040,10064,11088,12112,13136,1
       4160,15184,16208,9168,10192,11216,12240,1326
       4,14288
20050  DATA  15312,16336
```

9.9 COMBINING PAGES 1 AND 2

When there are two different pictures in pages 1 and 2 of high resolution which are related in some way, combining them to become one (superimposing them) is not a straightforward matter. Yet, at this moment we have the tools to do it. As usual, subroutine HSCR will play a basic role.

Program COMBINE starts by setting LOMEM: 24575 to protect the two pages of high resolution. SCND gets the distance between pages 1 and 2, and D is set to click the speaker.

Line 10 PRINTs the name of the program so that the screen is not blank while the initialization takes place. Lines 20 and 40 DIMension and READ M% ( ), the addresses of the beginning of every line of page one, adding SCND, so that the addresses become those of the second page. Line 50 displays the following menu:

1-LOAD PICTURE 1
2-LOAD PICTURE 2
3-COMBINE

1-LOAD PICTURE 1

Line 80 sets X$ to 2 and jumps to subroutine 800, which asks for a filename and returns to the menu if answered with an empty string. If a name is provided, line 810 BLOADs the file, using X$ to indicate the page into which it is to be loaded.

Since in hexadecimal the addresses of the pages are $2000 and $4000 (much easier to remember than 8192 and 16384), the same line BLOADs both pages 1 and 2, using X$ to place them where they are wanted.

Page one has been chosen to be the one in which the combination will be made.

2-LOAD PICTURE 2

Same as load picture 1, into page 2.

3-COMBINE

Lines 170 through 200 set page one of high resolution graphics with no text. Line 210 sets HCOLOR to white, and the activity switch to page one.

Line 500 starts the X and Y loops that will scan every point in page two. Line 510 calls subroutine 1000 (HSCR), which has been studied previously (see program BIT SCANNER), and which returns in the boolean variable HSCR whether the point in X, Y is ON or OFF.

If it is ON, HPLOT draws it in page one. Notice that the same X and Y were used to scan and HPLOT; since M% ( ) was set to the addresses of the second page of high resolution, the pair X, Y scan page two, and the corresponding points are HPLOTted in page one. Since it is assumed that there are more black points than white, having set the address in M% ( ) to point to the second page saves adding SCND every time a point is scanned by HSCR; the few (that is the supposition) that are found ON are drawn directly in page one, without any subtraction of addresses.

Line 520 emits a click every time a line is finished. Line 530 gives you the option of viewing the picture again, or BSAVEing it.

```
1  REM  ** COMBINE **
5  LOMEM: 24575:SCND = 8192:D =  - 16336
10  HOME : INVERSE : HTAB 13: PRINT "PAGE COMBINE
    ": NORMAL
20 D$ =  CHR$ (4): DIM M%(191): FOR I = 0 TO 191
40  READ M%(I):M%(I) = M%(I) + SCND: NEXT
50  VTAB 10: PRINT "1-LOAD PICTURE 1": VTAB 12: PRINT
    "2-LOAD PICTURE 2"
55  VTAB 14: PRINT "3-COMBINE ";: CALL  - 958
60  GET W$:V =  VAL (W$): IF V < 1 OR V > 3 THEN
    60
70  ON V GOTO 80,90,170
80 X$ = "2": GOSUB 800: GOTO 50
90 X$ = "4": GOSUB 800: GOTO 55
170  POKE  - 16304,0
180  POKE  - 16302,0
190  POKE  - 16300,0
200  POKE  - 16297,0
210  HCOLOR= 3: POKE 230,32
```

```
500   FOR Y = 0 TO 191: FOR X = 0 TO 279
510   GOSUB 10000: IF HSCR THEN  HPLOT X,Y
520   NEXT :W =  PEEK (D) +  PEEK (D): NEXT : TEXT
      : HOME : END
800   VTAB 16: HTAB 3: INPUT "FILENAME:";FL$: IF F
      L$ = "" THEN  POP : GOTO 50
810   PRINT : PRINT D$"BLOAD "FL$",A$"X$"000": RETURN
10000 BYTE =  INT (X / 7):BIT = (X - BYTE * 7) +1
10010  IF PBYTE = BYTE THEN 10070
10015 N =  PEEK (M%(Y) + BYTE)
10016  IF N = 0 THEN HSCR = 0:X = X + 6: RETURN
10017 A$ = ""
10020  FOR I = 7 TO 0 STEP  - 1
10030 DI =  INT (N / 2 ^ I)
10040 A$ =  STR$ (DI) + A$
10050 N = N -  INT (DI * 2 ^ I)
10060  NEXT
10065 PBYTE = BYTE
10070  IF  MID$ (A$,BIT,1) = "1" THEN HSCR = 1: RETURN
10080 HSCR = 0: RETURN
20000  DATA  8192,9216,10240,11264,12288,13312,14
      336,15360,8320,9344,10368,11392,12416,13440,
      14464,15488,8448,9472,10496,11520,12544,1356
      8,14592,15616,8576,9600,10624,11648,12672,13
      696,14720,15744,8704,9728,10752,11776,12800,
      13824
20010  DATA  14848,15872,8832,9856,10880,11904,12
      928,13952,14976,16000,8960,9984,11008,12032,
      13056,14080,15104,16128,9088,10112,11136,121
      60,13184,14208,15232,16256,8232,9256,10280,1
      1304,12328,13352,14376,15400,8360,9384,10408
      ,11432
20020  DATA  12456,13480,14504,15528,8488,9512,10
      536,11560,12584,13608,14632,15656,8616,9640,
      10664,11688,12712,13736,14760,15784,8744,976
      8,10792,11816,12840,13864,14888,15912,8872,9
      896,10920,11944,12968,13992,15016,16040,9000
      ,10024
20030  DATA  11048,12072,13096,14120,15144,16168,
      9128,10152,11176,12200,13224,14248,15272,162
      96,8272,9296,10320,11344,12368,13392,14416,1
      5440,8400,9424,10448,11472,12496,13520,14544
      ,15568,8528,9552,10576,11600,12624,13648,146
      72,15696
20040  DATA  8656,9680,10704,11728,12752,13776,14
      800,15824,8784,9808,10832,11856,12880,13904,
      14928,15952,8912,9936,10960,11984,13008,1403
      2,15056,16080,9040,10064,11088,12112,13136,1
      4160,15184,16208,9168,10192,11216,12240,1326
      4,14288
20050  DATA  15312,16336
```

9.10 ANIMATION

To make an object move smoothly on the screen, it is necessary to do all the erasing and drawing so that it does not show on the screen. One finished page is displayed while all the activity takes place backstage, in the other one.

When everything is ready, the pages are flipped and work starts in the one that is now invisible.

Before doing any drawing, it is necessary to erase the objects that will change position (an object that will not move does not need to be touched). I prefer to erase, redrawing the shapes with black rather than using the XDRAW command.

To erase a shape, its position in the screen must be remembered, or else the entire screen will have to be erased, slowing the process incredibly. After the erasing is completed, the drawing can be started. When the screen is ready, the pages are flipped, and the activity switch is reversed, so that the changes occur in the opposite page.

Program ANIMATION is a very simple example of objects moving on the screen. It expects a shape table to be at position 25000, with at least two shapes in it. (See SHAPE GENERATOR, 9.14.)

Line 10 POKES the shape address in locations 232 and 233. Line 15 initializes the following variables:

ZE = 0
AN = 1
TW = 2
P1 = –16300 (the switch for page one)
P2 = –16299 (the switch for page two)
WS = 230 (the activity switch)
S1 = 32 (when POKEd into WS, the drawing goes to page one)
S2 = 64 (drawing to page two)

Line 27 sets HGR 2 (a fast way to set full-page high resolution and clear the screen at the same time), the HCOLOR and attributes for the shapes are set, the activity switch is directed toward page one, and it is cleared. Page one is displayed and the activity switch is directed to page two.

Line 30 starts the I-loop that will be used to determine the position of the shapes. Line 40 toggles F (it can be 1 or 2). Line 60 erases the shapes in this page using X1 (F), Y1 (F), which stores the previous position of shape one in this page, and X2 (F), Y2 (F), which keeps the previous position of shape two in this page. HCOLOR is set back to white.

Line 70 calculates the new coordinates for the shapes (number one moves diagonally starting at the upper left corner going down and right, and number two does the same trajectory backwards) and draws them. Depending on F (the page flag), lines 90 and 100 flip to the opposite page, and change the activity switch.

```
1  REM  ** ANIMATION **
10  HIMEM: 25000: POKE 232,168: POKE 233,97
15 ZE = 0:AN = 1:TW = 2:P1 =  - 16300:P2 =  - 162
     99:WS = 230:S1 = 32:S2 = 64
27  HGR2 : HCOLOR= 3: ROT= 0: SCALE= 1: POKE WS,S
     1: CALL 62450: POKE P1,ZE: POKE WS,S2
30  FOR I = 44 TO 191
40 F = F + 1: IF F = 3 THEN F = 1
60  HCOLOR= 0: DRAW AN AT X1(F),Y1(F): DRAW TW AT
     X2(F),Y2(F): HCOLOR= 3
70 X1(F) = I:Y1(F) = I: DRAW AN AT I,I:X2(F) = 19
     1 - I:Y2(F) = X2(F): DRAW TW AT X2(F),Y2(F)
80  ON F GOTO 90,100
90  POKE P2,ZE: POKE WS,S1: GOTO 110
100  POKE P1,ZE: POKE WS,S2
110  NEXT
120  GOTO 30
```

9.11 SHAPES

The commands available for shape manipulation are invaluable tools for high resolution graphics; instead of having to draw a shape dot by dot or line by line, these instructions draw a predefined picture at machine language speed and let you scale up or rotate the figure, which is otherwise very hard and slow to do.

Unfortunately, the creation of the shapes is a complicated process. It is very frustrating to spend long minutes or hours trying to assemble a shape table and, when drawing the shape on the screen, to discover that it has nothing to do with the intended one.

Fortunately (there is always a good side), the process can be left for the computer to handle, if after understanding exactly how shapes work, we devise a program to do all the hard work for us.

9.12 CREATION OF A SHAPE TABLE

The first thing that must be done to create a shape is to transfer the drawing to graph paper, reducing it to squares that will be drawn, and leaving all the rest blank. Figure 9.1 shows how a rectangle with a dot inside would look.

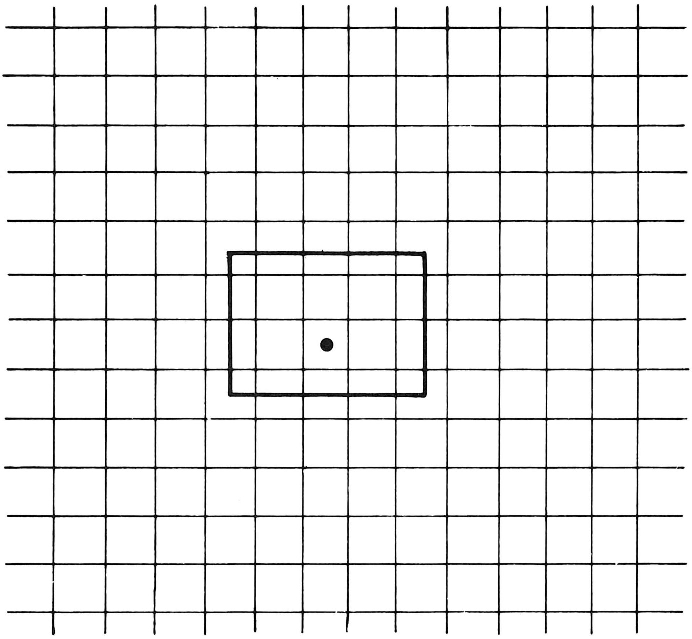


Figure 9.1. Intended drawing.

You must decide where the starting point will be. In the figure, the point in the top left corner was picked as the beginning of the shape.

The second step is to decide upon a path that is short and that at the same time covers the entire picture. To move from one square to another, only horizontal or vertical movements are allowed. Table 9.2 shows the decimal and binary codes for the eight possible movements.

Table 9.2. Codes for the shape movements.

DECIMAL	BINARY	ACTION
0	000	Move up without drawing
1	001	Move right without drawing
2	010	Move down without drawing
3	011	Move left without drawing
4	100	Move up drawing
5	101	Move right drawing
6	110	Move down drawing
7	111	Move left drawing

Now you draw on a separate piece of paper the trajectory, indicating which movements draw and which do not. Figure 9.2 shows the path with arrows. The convention adopted here is that if the arrow is crossed, it draws; otherwise it does not.

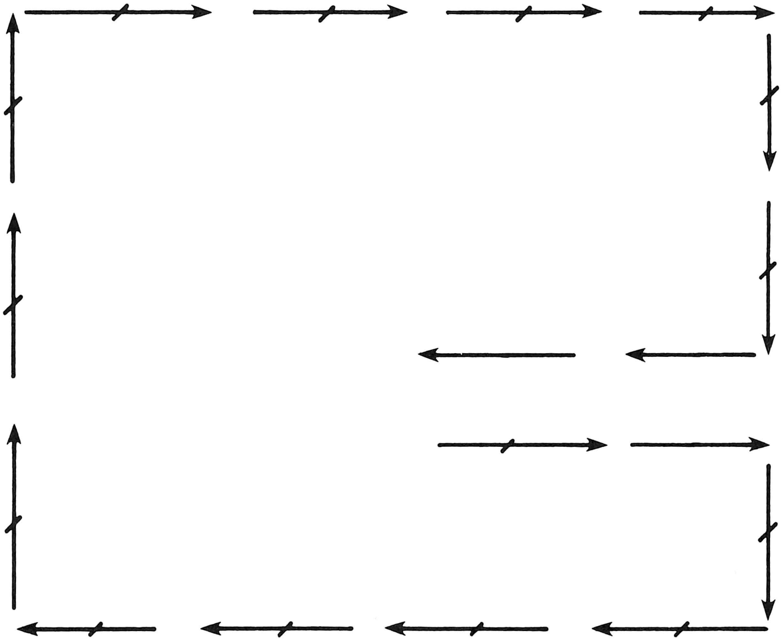
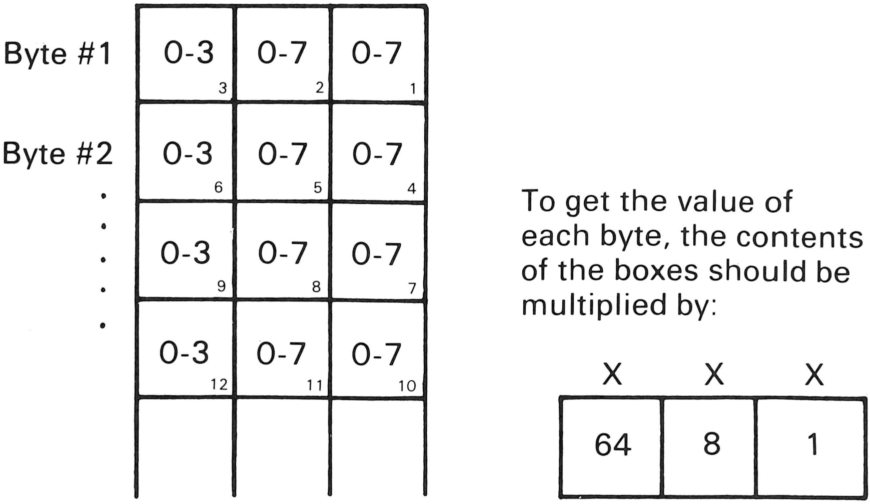


Figure 9.2. Selected path.

Every byte in the shape table is divided into three sections, as shown in figure 9.3. The two sections to the right can store



To get the value of each byte, the contents of the boxes should be multiplied by:

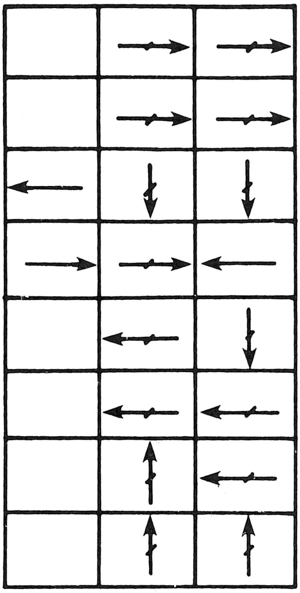
Figure 9.3. Byte sections for a shape.

numbers from 0 to 7 (binary 000 to 111); the left one can only store values from 0 to 3 (binary 00 to 11). This means that the left section cannot store an arrow that draws. Every 0 section to the left of the byte is totally ignored; if the three sections are 0, Applesoft assumes that the end of the shape has been reached.

Starting with the right hand section of the first byte, the code for the first arrow is found and put there. The second arrow goes into the middle section of the byte (as shown in Figure 9.4). Since the third arrow must draw and the left section of a byte cannot store such a number, it is left as 0 and this third number is placed in the right hand section of the second byte. The process is repeated until all the arrows are in the corresponding bytes; the left one is set to 0.

When all the bytes of the shape have been calculated, the only thing missing before actually moving them into the table is its heading.

The first byte of the table (byte # 0) contains the number of shapes in it (see Figure 9.5). The second byte is not used, and starting in the second byte from the start of the table, the distances of every shape to the beginning of the table are placed in two bytes: the least significant one is first, and the most significant follows.



×64	×6	×1		
0	5	5	=	45
0	5	5	=	45
3	6	6	=	246
1	5	3	=	107
0	7	6	=	62
0	7	7	=	63
0	4	7	=	39
0	4	4	=	36

Figure 9.4. Translation of arrows into bytes.

Start →	0	1	← Number of shapes
	1	0	
	2	4	← Distance from start to first byte of the table
	3	0	
	4	45	← Beginning of shape
	5	45	
	6	246	
	7	107	
	8	62	
	9	63	
	10	39	
	11	36	
	12	0	← End of shape

Figure 9.5. Final shape table.

Once the table is ready, it must be POKEd into memory, in an area that will not be disturbed by the program. If the shape is not too big, it can be put in the small free area starting in location 768, but since we use that same area for the sound routine and for other purposes, it is better to set HIMEM to an address below DOS (see appendix D of the DOS manual for the lowest DOS address in different memory systems) that give us room to put it there.

Once the table is ready, two pointers must be set before using it. Locations 232 and 233 should have the byte representation of the location of the shape table in memory. In the example, since HIMEM was set to 25000, the two numbers POKEd are 168 in location 232, and 97 in 233.

9.13 USING SHAPES

Once the shape table is in memory, the following commands are available (high resolution must be set):

ROT = 16

Sets the rotation of the shape with respect to its original position. Even though values from 0 to 255 can be used as arguments, the full 360 degree range can be handled with values from 0 to 64. An increase of 8 units changes the angle of the shape 45 degrees.

If SCALE is set to 1, only eight values produce a real rotation (in this case, ROT = 3, for example, would be taken as ROT = 0). When scale is set to 2, the number of active rotations goes up to eight, etc.

SCALE = 1

Sets the size at which the shape is drawn. SCALE = 1 is the smallest and the size is multiplied by two with every unit increased. Following the pattern, SCALE = 255 should be the biggest, however SCALE = 0 is actually the biggest, since it is equivalent to SCALE = 256.

DRAW I AT X, Y

Draws shape number I at position X, Y, with the last HCOLOR, ROT, and SCALE set.

XDRAW I AT X, Y

Draws shape I with the complement of the color with which it was originally drawn, using the current SCALE and ROT. Its effect is to erase shape I.

Even with the method described here, in which there is no need to deal with binary numbers, a great possibility of error exists when assembling a shape table.

One special case must be considered: When you have an arrow moving up and not drawing (code 0), it will only be understood by Applesoft if it has some other nonzero number to its left.

Otherwise, it will be ignored and, if three of these are in the same byte (producing a 0 byte, even though originally it was meant to be three arrows moving up), the shape will terminate.

This can be avoided by choosing a different trajectory or by adding a couple of arrows moving left and right, which do nothing to the actual position of the cursor, but add two nonzero values to the byte, fixing the problem. See Figure 9.6.

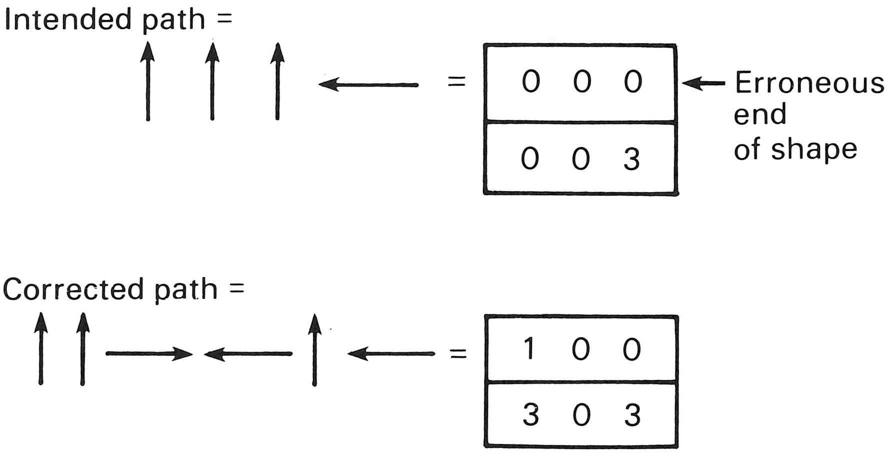


Figure 9.6. Example of anomalous shape.

9.14 SHAPE GENERATOR

To avoid all the work described in the previous section, program SHAPE GENERATOR lets you draw the shape in a way similar to the high resolution editor.

The usual commands are available for cursor movement:

I - to go up
J - to go left
K - to go right
M - to go down

"D" is used to toggle between DRAW and DON'T DRAW modes.

Line 5 sets X to 100 and Y to 50. The shape will be started in the middle of the screen to make the work easier. U% ( ) will store the values of the bytes of the shape; M% ( ) will keep the code of every one of the arrows to redraw it when it is necessary. BL will be used to click the speaker.

Line 10 sets ZE to 0, AN to 1, TH to 3, TN to 10, KB to the address where the characters from the keyboard are read, KR to the address where the keyboard strobe is reset, and TRUE and FALSE to their Applesoft equivalents.

Line 12 asks for the initial address, which will be the place in memory where the shape table will start, i.e., where the number of shapes in the table will reside. It then asks if more shapes are in that address, because the creation of the first is a bit different from the others. Line 13 sets FI (a boolean variable indicating that no shapes are in memory) to TRUE.

Line 14 asks if HIMEM should be set, and line 15 does so if answered YES. It also POKEs into locations 232 and 233 (the locations where Applesoft looks for the start of the table) the initial address, and POKEs 0 into the location IA, indicating 0 shapes are in the table.

Line 20 sets D$ to CHR$ (4) for the disk operations, because if HIMEM is lower than where it had been stored originally, its contents will be lost.

Line 33 shows the following menu:

1-CREATE A NEW SHAPE
2-SAVE TO DISK
3-LOAD FROM DISK
4-END

1-CREATE A NEW SHAPE

Line 40 initializes the following variables to zero:

L - Will keep track of the section in the current byte.
T - Will store temporarily the value of the current byte until it is complete.
U - Will be used as the index for U% ( ), the bytes of the table.

M - Will be used as the index for M% ( ), the codes for every arrow entered.

X = FRE (8) performs a garbage collection every time a new shape will be entered, to avoid delays in the middle of the process. X and Y are set to the point at the middle of the screen.

Page one of high resolution with four lines of text is set, HCOLOR is set to three (white), the text area is cleared, and control is transferred to line 200, where D (the DRAW-DON'T DRAW indicator) is toggled and its condition PRINTed in line 23. K is the blinking cursor counter: It is incremented by 10 so that as soon as the high resolution page is displayed, the cursor appears. Line 110 is the beginning of the input cycle.

Subroutine 2000 reads the keyboard; if some key has been pressed, it resets the keyboard strobe, assigns to W$ the CHR$ of the character typed, and RETURNs. Otherwise, line 2020 increments K; if it is still lower than 5, it goes back to 2000. Otherwise, lines 2030 and 2040 toggle the cursor's HCOLOR, HPLOT it, and RETURN.

If either I, J, K, or M have been pressed, lines 120 through 180 set F to the code of the corresponding movement (up, left, right, or down) and call subroutine 1000; on return from it the value of X or Y, depending on the case, is updated; and control goes back to line 110.

Subroutine 1000 does the actual conversion of the movement into the corresponding code.

The difference between the codes for DRAW and DON'T DRAW is exactly four for all four possibilities. Therefore, line 1000 multiplies D-1 by 4 to add it to the actual code in F.

If D is in the DON'T DRAW condition (value 1), 0 is added to F; if it is 2 (draw), 4 is added to F; the result is stored in G. M% (M) gets the resulting code and M is incremented so that the next time G ends up in the next element of M% ( ).

Line 1010 sets the color to (D–1)✱3, that is, 0 if D is 1 and 3 if it is 2. With that HCOLOR, a point is HPLOTted in X, Y.

Notice that if the HCOLOR is black (DON'T DRAW) and the cursor is over a point that had previously been painted, it will erase it, which will not happen in the actual shape, since the DON'T DRAW mode does not mean to erase, simply to move.

This will cause the shape on the screen to be different from the one that is actually going to be produced; to fix this, instead of using the function HSCR, which would slow down the process (but which would work), the option is given which will be explained when lines 250 through 350 are described.

When converting this resulting code (G) into its byte equivalent, the following cases must be considered:

1. First section: Line 1049 adds G (multiplied by the section factor) to T, and increments L, to indicate that the current section is number two.
2. Second section: If G is 0, there is a great chance that the third section is 0, since it cannot store values greater than 3. To avoid this, line 1040 adds a move-left arrow code in the third section (adding 192, equivalent to 3 {move left} multiplied by 64 {the factor for the third section}). The resulting value (T) is stored in U% (U), U is incremented, and T is assigned 1 (move right without drawing in the first section). That way, the shape has to turn left and right without drawing, not affecting the resulting picture, but avoiding the problem of the ignored 0 at the left of the byte, which distorts the picture.

   If G is not 0, line 1049 multiplies it by 8^ L, the section factor, and adds it to T. Since this line 1049 is used when L is 0 also, the subroutine RETURNs if L is less than 2. Otherwise, line 1050 assigns the finished byte to U% (U), increments U, sets T to 0 (the byte is now empty), sets L to 0 (first section), and RETURNs.
3. Third section: Line 1030 adds the resulting code to T only if G is either 1, 2, or 3, the only values usable in the third section, and goes to line 1050 which updates everything just as it did for the second section. If G is not usable in this section, T is assigned to U% (U), U is incremented, T is set to G since it could not be put in T previously, and L is set to 1 (second section).

If D is typed, lines 200 and 210 toggle D (from DRAW to DON'T DRAW or vice versa), and PRINTs the corresponding message.

If ctrl-shift-M is pressed, line 250 sets HGR, erasing the picture in memory. X and Y are set to 100 and 50, their HOME for this program, and HCOLOR is set to 3 (white).

Line 260 starts the I-loop that will process M%( ) for as many arrows or cursor movements as have occurred so far (until M−1). Line 270 stores M% (I) into W. If it is greater than or equal to 4 (DRAW), it subtracts 4 (to leave only the move code), and HPLOTs X, Y. If W is a DON'T DRAW code, nothing is done to it, and line 290 makes both cases update X or Y, depending on the code. That way, the shape is reproduced exactly as it will be DRAWn when used.

If <esc> is pressed, U% (U) is assigned the last T if it isn't zero. N gets the current value of U (the number of bytes in this shape). "PROCESSING" is PRINTed on the TEXT screen, and the actual setting of the shape table starts.

If FI is TRUE, this is the first shape, and line 400 sets it to FALSE, so that the following shapes are not processed as the first one. 1 is POKEd into IA, the initial address (there is 1 shape in the table); 0 is POKEd into location IA+1.

In IA+2 and IA+3, the locations where the distance between the beginning of the table and the beginning of shape 1 must be, 4 and 0 respectively are POKEd (the first byte of shape 1 is going to be in location IA+4). Then, all U% ( ) is POKEd starting at IA+4. After the last byte, 0 is POKEd, indicating the end of the shape.

If the shape is not the first one, line 408 puts in NS the current number of shapes on the table. LA gets the position of the first byte where the distance from IA to the last shape resides (since every distant pair takes two bytes, NS✱2 indicates how many bytes are between IA and the last shape pointer). LS gets the actual address in memory where the last shape begins.

Line 420 looks for the next 0 in memory, indicating the end of the shape. If it does not find it, line 430 increments I and returns to 420.

Line 440 moves every byte in the actual shape table (excluding the distance indicatives at the beginning) two places, to make room for the pair of distance bytes of the shape that is being inserted. Line 450 sets NA to the distance between IA and the place where the new shape will start. MS and LS get the most and least significant bytes of this number, and POKEs them at the end of the distance part of the shape table.

Lines 460 and 470 add 2 to every low significant byte of the distance part of the table, to account for the movement of the actual shape part. If the resulting value is greater than 255 and therefore cannot be POKEd into a byte, the most significant byte is increased by one, and 256 is subtracted from the least significant byte.

Line 480 POKEs the actual new table starting at IA+NA (new address). A 0 is POKEd at the end, and line 490 updates the number of tables in IA to its previous value plus one.

2-SAVE

If line 50 finds no shapes to SAVE, it returns to the menu. Otherwise, line 55 finds and puts into LA the distance between IA and the beginning of the last shape. An I-loop looks for the end of that last shape indicated by a 0, so that it can determine exactly the length to BSAVE. Line 65 calls subroutine 700 which asks for a filename and returns to the menu if answered with a null string; otherwise, the entire table is BSAVEd.

3-BLOAD

Line 75 calls subroutine 700 for a filename, and BLOADs the shape, starting at IA. That way, a shape table can be relocated easily. FI is set to FALSE to indicate at least one shape is on the table (an empty shape table being BLOADed is illegal here).

```
1  REM  ** SHAPE GENERATOR **
5  LOMEM: 16384:X = 100:Y = 50: DIM U%(2000),M%(2
     000):BL =  - 16336
10 ZE = 0:AN = 1:TH = 3:TN = 10:KB =  - 16384:KR =
      - 16368:TRUE = 1:FALSE = 0
12  TEXT : HOME : VTAB 8: INPUT "INITIAL ADDRESS:
     ";IA: PRINT : PRINT "ARE THERE MORE SHAPES I
     N THIS ADDRESS ? ": PRINT  TAB( 20);"(Y/N) "
     ;
13  GET W$: IF W$ = "N" THEN FI = TRUE
14  PRINT : PRINT "1-SET HIMEM:": PRINT : PRINT "
     2-CONTINUE ";
15  GET W$: IF W$ = "1" THEN  HIMEM: IA:S =  INT
     (IA / 256): POKE 232,IA - S * 256: POKE 233,
     S: POKE IA,0
20 D$ =  CHR$ (4)
30  TEXT : HOME : VTAB 2: HTAB 6: PRINT "THERE AR
     E " PEEK (IA)" SHAPES IN MEMORY"
33  VTAB 11: PRINT "1-CREATE A NEW SHAPE": PRINT
     : PRINT "2-SAVE TO DISK": PRINT : PRINT "3-L
     OAD FORM DISK": PRINT : PRINT "4-END ";
```

```
35  GET W$: IF W$ < "1" OR W$ > "5" THEN 35
38  ON  VAL (W$) GOTO 40,50,75,80
40 L = 0:T = 0:U = 0:M = 0:X =  FRE (8):X = 100:Y
     = 50: HGR : HOME : HCOLOR= 3:D = 1: GOTO 20
    0
50 S =  PEEK (IA): IF S = 0 THEN 30
55 LA =  PEEK (IA + S * 2) +  PEEK (IA + S * 2 +
    1) * 256: FOR I = LA + IA TO LA + IA + 200: IF
     PEEK (I) = 0 THEN L = I - IA: GOTO 65
60  NEXT : GOTO 30
65  GOSUB 700: PRINT : PRINT D$"BSAVE "FL$",A"IA"
    ,L"I + 1 - IA: GOTO 30
75  GOSUB 700: PRINT : PRINT D$"BLOAD "FL$:FI = F
    ALSE: GOTO 30
80  HOME : END
100  HGR : HCOLOR= 3
110  GOSUB 2000: IF W$ <  > "I" THEN 130
120  GOSUB 5000:F = 0: GOSUB 1000:Y = Y - 1: GOTO
     110
130  IF W$ <  > "M" THEN 150
140  GOSUB 5000:F = 2: GOSUB 1000:Y = Y + 1: GOTO
     110
150  IF W$ <  > "J" THEN 170
160  GOSUB 5000:F = 3: GOSUB 1000:X = X - 1: GOTO
     110
170  IF W$ <  > "K" THEN 190
180  GOSUB 5000:F = 1: GOSUB 1000:X = X + 1: GOTO
     110
190  IF W$ <  > "D" THEN 220
200 K = K + TN: IF D = 1 THEN D = 2: VTAB 23: HTAB
     3: PRINT "DRAW        ";: GOTO 110
210 D = 1: VTAB 23: HTAB 3: PRINT "DON'T DRAW";: GOTO
     110
220  IF W$ <  >  CHR$ (29) THEN 240
230 U%(U) = T: GOTO 30
240  IF W$ <  >  CHR$ (20) THEN 360
250  HGR :X = 100:Y = 50: HCOLOR= 3
260  FOR I = 0 TO M - 1
270 W = M%(I): IF W >  = 4 THEN W = W - 4: HPLOT
     X,Y
290  ON W + 1 GOTO 300,310,320,330
300 Y = Y - 1: GOTO 340
310 X = X + 1: GOTO 340
320 Y = Y + 1: GOTO 340
330 X = X - 1
340  NEXT
350  GOTO 110
360  IF W$ <  >  CHR$ (27) THEN 110
370 U%(U) = T:N = U: TEXT : HOME
380  VTAB 13: HTAB 16: INVERSE : PRINT "PROCESSIN
     G": NORMAL
```

```
400  IF FI THEN FI = FALSE: POKE IA,1: POKE IA +
     1,0: POKE IA + 2,4: POKE IA + 3,0: FOR I = 0
      TO N: POKE IA + 4 + I,U%(I): NEXT : POKE IA
      + 4 + I,0: GOTO 30
408 NS =  PEEK (IA):LA = IA + NS * 2:LS = IA +  PEEK
     (LA) +  PEEK (LA + 1) * 256
410 I = LS
420  IF  PEEK (I) = 0 THEN 440
430 I = I + 1: GOTO 420
440  FOR J = I + 2 TO LA + 2 STEP  - 1: POKE J, PEEK
     (J - 2): NEXT
450 NA = I - IA + 3:MS =  INT (NA / 256):LS = NA -
     MS * 256: POKE LA + 2,LS: POKE LA + 3,MS
460  FOR J = 2 TO LA - IA STEP 2:LS =  PEEK (IA +
     J):MS =  PEEK (IA + J + 1):LS = LS + 2: IF L
     S >  = 256 THEN MS = MS + 1:LS = LS - 256
470  POKE IA + J,LS: POKE IA + J + 1,MS: NEXT
480  FOR J = 0 TO N: POKE IA + NA + J,U%(J): NEXT
     : POKE IA + NA + J,0
490  POKE IA,NS + 1: GOTO 30
700  PRINT : PRINT : INPUT "  FILENAME:";FL$: IF
     FL$ = "" THEN  POP : GOTO 30
710  RETURN
1000 G = (D - 1) * 4 + F:M%(M) = G:M = M + 1
1010  HCOLOR= (D - 1) * 3: HPLOT X,Y:K = 5
1020  IF L < 2 THEN 1040
1030  IF G > 0 AND G < 4 THEN T = T + 8 ^ L * G: GOTO
     1050
1035 U%(U) = T:U = U + 1:T = G:L = 1: RETURN
1040  IF L = 1 AND G = 0 THEN U%(U) = T + 192:U =
     U + 1:T = 1:L = 1: RETURN
1049 T = T + 8 ^ L * G: IF L < 2 THEN L = L + 1: RETURN
1050 U%(U) = T:U = U + 1:T = 0:L = 0: RETURN
2000 W$ = "":W =  PEEK (KB): IF W < 128 THEN 2020
2010  POKE KR,ZE:W$ =  CHR$ (W - 128): RETURN
2020 K = K + 1: IF K < 5 THEN  RETURN
2030 K = 0: IF C = 0 THEN C = 1: HCOLOR= TH: HPLOT
     X,Y: RETURN
2040 C = 0: HCOLOR= ZE: HPLOT X,Y: RETURN
5000 WW =  PEEK (BL) +  PEEK (BL) +  PEEK (BL): RETURN
```

9.15 TEXT IN HIGH RESOLUTION

A big limitation of both graphic modes is that it is impossible to display text and graphics simultaneously. And many times it is not only convenient but indispensable.

The characters in the Apple are formed in a matrix of 5 * 7 dots with one more on each side to separate them, and one above the character. These dots are the same size as the high resolution ones.

Using the shape generator, an exact copy of each character can be made, a whole set of lower case letters can be added, and there is still room for some extra characters.

Before you run the program in this section you will need a shape table called ALPHABET, available in the disk; if you did not buy the disk, the program ALPHABET CREATOR at the end of this chapter will create it for you. The same applies to ALPHABET 70 (the shape table for the program in the next section), which is generated by the program ALPHABET 70 CREATOR.

In the shape table provided, the first character is "!", and the shape number corresponds to its ASCII minus 32. The same holds for all the characters.

Having this shape table in memory, it suffices to DRAW a character in a certain position using ROT = 0 and SCALE = 1. The character displayed will have absolutely no difference from the one in TEXT mode.

To make things a little easier, program FORTY CHARACTERS allows you to enter the characters in the same way as when you are in TEXT, combining it with graphics, and BSAVEing it for future use.

Ctrl-A is used to switch between the three modes: upper case, lower case, and special characters.

The cursor is not a blinking rectangle but an underscore (CHR$ (95)).

Each one of the three modes is recognized by the height of the cursor: when it is in the bottom, it draws upper case; in the middle, lower case; and in the top, the special characters set.

Certain characters usually not available from the Apple keyboard have been incorporated.

The program begins setting HIMEM to 25000. This is a completely arbitrary number, and any other can be used, taking into consideration that the first page of high resolution is needed.

Line 10 BLOADs the file ALPHABET. The address of the table is POKEd into locations 232 and 233, POKE 230, 32 sets the activity switch to page one, and the HCOLOR is set to 3 (white).

Line 20 sets ROT to 0 and SCALE to 1, BL$ to the bell character, BL to the address to click the speaker, and the following menu is displayed:

1-RETURN TO HIGH RESOLUTION
2-SAVE
3-LOAD
4-CLEAR SCREEN
5-TOGGLE CURSOR
6-MOVE CURSOR
7-QUIT

1-RETURN TO HIGH RESOLUTION

To allow the simultaneous use of text and graphics, the program does not enter high resolution with HGR, because any previous picture would be erased.

Line 3000 switches to text (at the beginning it is not necessary, but every time the program returns to the menu we do not want to have to redraw a picture that is intact) executing a POKE −16303, 0. The cursor is sent to the end of the text and the call −958 clears the screen from the cursor to the bottom end.

Line 3030 sets page one of high resolution graphics with no text, and jumps to line 1000, the input cycle.

Subroutine 2000, called by 1000, RETURNs if F1 is TRUE, which indicates the cursor is OFF. In line 2002, if X and Y are found in the bottom right of the screen, PX is set to that address and the subroutine RETURNs, since there can be no cursor beyond the last character of the screen. In line 2005, PX is set to the dot equivalent of X, the underline (shape # 117) is drawn, and the subroutine RETURNs.

Line 1000 GETs W$, the next character, and calls subroutine 2010, which RETURNs if the cursor is OFF (F1 is TRUE in line 2010), or if it is already at the last position of the screen (line 2012). Otherwise, line 2015 erases the cursor and returns.

If line 1000 finds that the character typed is either a space or a right arrow (CHR$ (21)), it calls subroutine 2300 to erase the cursor or the character at the cursor position, by drawing shape 118 (a block the size of a character) with black. A jump to line 1130 updates the cursor to the next position. If X is not in the rightmost location (39), X is incremented; if X is at 39, line 1140 beeps if it is in the bottom line, and RETURNs. Otherwise, line 1145 sets X to 0 (left margin) and Y to Y+1, the next line.

If the character pressed is <return>, line 1005 jumps to 1140 to send the cursor to the next line and the leftmost column, providing the cursor position is not the right-bottom corner.

If the character pressed is ctrl-J (CHR$ (10) or line feed), lines 1007 and 1008 increment Y (next line without carriage return), if the cursor is not in the bottom line.

Line 1010 takes care of the very unlikely ctrl-@, which passes over the GET instruction and sets W$ to " ".

If the backspace is typed (CHR$ (8)), line 1012 decrements X if it is not 0, and calls subroutine 2020. If PX (the previous X) is greater than 0, NX (new X) is set to PX, since the cursor must backspace. If X is 0, line 1013 beeps if Y indicates the cursor is at the top line and no backspacing is possible. Otherwise, line 1014 decrements Y and sets X to 39, the right hand margin of the screen. Subroutine 2020 finds the right position for the new X and new Y (NX, NY) and erases any character in that position by drawing shape 118 with black.

If ctrl-A is pressed, lines 1030 and 1040 set FL (the shift mode: upper case, lower case, or special symbols) to the next value in the permutation cycle 1, 2, 3, back to 1, etc.

If ctrl-E is pressed (CHR$ (5)), lines 1043 and 1044 decrement Y if possible, to move the cursor up.

If <esc> is pressed, line 1047 jumps to line 3000, which calls the menu.

If none of the above is detected, it means that the character might be a displayable one, and line 1050 sets S to the ASCII code of the character minus 32, which corresponds to the shape number. This shape table can be considered as a shift of the ASCII code, designed that way because no control characters are needed.

If FL is 1 (upper case mode), line 1090 draws the corresponding character (including the usual symbols like !, ", #, etc., and the numbers) if it is displayable, or beeps otherwise.

If FL is 2 (lower case), line 1060 selects the letters; if it is one, it draws the shape adding 32 to S (there is an offset of 32 in the ASCII code between upper and lower case). If it is not a letter, line 1070 checks to see if it is a number or a standard symbol; if it is, it draws it. Otherwise it beeps and jumps to 1000.

If FL is in the special character mode, line 1110 beeps if W$ doesn't correspond to a letter: All the 26 symbols are drawn when a letter is pressed.

2-SAVE

Line 3040 calls subroutine 3200 which asks for a filename and returns to the menu if answered with an empty string; otherwise, page one is BSAVEd.

3-LOAD

Line 3050 calls subroutine 3200 for a filename and BLOADs a picture if the filename is provided.

4-CLEAR SCREEN

Line 3060 asks for confirmation (to prevent possible mistakes) and, if "Y" is pressed, it executes a CALL 62450, the machine language routine, to clear the screen.

5-TOGGLE CURSOR

Line 3080 changes the state of F1, turning OFF the cursor if it is ON, and vice versa. When the cursor is OFF, the characters are not erased as new ones are DRAWn; this way, more than one character can be in the same place at the same time.

Line 2090 PRINTs the new state, and line 3100 waits, until a key is pressed, to return to the menu.

6-MOVE CURSOR

Line 3110 shows the current position of the cursor, and does the INPUT of its new coordinates. No check is made, so care must be taken to provide legal values.

```
1  REM  ** FORTY CHARACTERS **
10  HIMEM: 25000:D$ =  CHR$ (4): PRINT D$"BLOAD A
     LPHABET": POKE 232,168: POKE 233,97: POKE 23
     0,32: HCOLOR= 3
20  ROT= 0: SCALE= 1:BL$ =  CHR$ (7):FL = 1:BL =
      - 16336: HOME
25  VTAB 6: PRINT "1-RETURN TO HIGH-RES": PRINT :
      PRINT "2-SAVE TO DISK": PRINT : PRINT "3-LO
     AD FROM DISK": PRINT : PRINT "4-CLEAR SCREEN
     "
27  PRINT : PRINT "5-TOGGLE CURSOR": PRINT : PRINT
     "6-MOVE CURSOR": PRINT : PRINT "7-QUIT ";
30  GOTO 3000
```

```
1000  GOSUB 2000: GET W$: GOSUB 2010: IF W$ =  CHR$
     (32) OR W$ =  CHR$ (21) THEN  GOSUB 2300: GOTO
     1130
1005  IF W$ =  CHR$ (13) THEN 1140
1006  IF W$ <  >  CHR$ (10) THEN 1010
1007  IF Y >  = 23 THEN  PRINT BL$;: GOTO 1000
1008 Y = Y + 1: GOTO 1000
1010  IF W$ = "" THEN  PRINT BL$;: GOTO 1000
1011  IF W$ <  >  CHR$ (8) THEN 1020
1012  IF X > 0 THEN X = X - 1: GOSUB 2020: GOTO 1
     000
1013  IF Y = 0 THEN  PRINT BL$;: GOTO 1000
1014 Y = Y - 1:X = 39: GOSUB 2020: GOTO 1000
1020  IF W$ <  >  CHR$ (1) THEN 1042
1030 FL = FL + 1: IF FL = 4 THEN FL = 1
1040  GOTO 1000
1042  IF W$ <  >  CHR$ (94) THEN 1047
1043  IF Y = 0 THEN  PRINT BL$;: GOTO 1000
1044 Y = Y - 1: GOTO 1000
1047  IF W$ =  CHR$ (27) THEN 3000
1050 S =  ASC (W$) - 32: ON FL GOTO 1090,1060,111
     0
1060  IF S >  = 33 AND S <  = 58 THEN  DRAW S + 3
     2 AT X * 7,Y * 8: GOTO 1130
1070  IF S > 0 AND S < 63 THEN  DRAW S AT X * 7 +
     2,Y * 8: GOTO 1130
1080  PRINT BL$;: GOTO 1000
1090  IF S >  = 1 AND S <  = 63 THEN  DRAW S AT X
      * 7,Y * 8: GOTO 1130
1100  PRINT BL$;: GOTO 1000
1110  IF S < 33 OR S > 58 THEN  PRINT BL$;: GOTO
     1000
1120  DRAW S + 58 AT X * 7,Y * 8
1130  IF X < 39 THEN X = X + 1: GOTO 1000
1140  IF Y >  = 23 THEN  PRINT BL$;: GOTO 1000
1145 X = 0:Y = Y + 1: GOTO 1170
1150  PRINT BL$;: GOTO 1000
1170  GOTO 1000
2000  IF F1 THEN  RETURN
2002  IF X = 39 AND Y = 23 THEN PX = 273: RETURN
2005 PX = X * 7:PY = Y: DRAW 117 AT PX,Y * 8 + 7 -
     (FL - 1) * 3: RETURN
2010  IF F1 THEN  RETURN
2012  IF X = 39 AND Y = 23 THEN  RETURN
2015  HCOLOR= 0: DRAW 117 AT PX,PY * 8 + 7 - (FL -
     1) * 3: HCOLOR= 3: RETURN
2020  IF PX > 0 THEN NX = PX:NY = PY: GOTO 2220
2030  IF PY = 0 THEN  PRINT BL$;: POP : GOTO 1000

2040 NY = PY - 1:NX = 280
2220  HCOLOR= 0: DRAW 118 AT NX - 7,NY * 8 + 1: HCOLOR=
     3: RETURN
```

```
2300  HCOLOR= 0: DRAW 118 AT X * 7,Y * 8 + 1: HCOLOR=
     3: RETURN
3000  POKE  - 16303,0: VTAB 19: HTAB 10: CALL  -
     958
3010  GET W$: IF W$ < "1" OR W$ > "7" THEN 3010
3020  ON  VAL (W$) GOTO 3030,3040,3050,3060,3080,
     3110,3070
3030  POKE  - 16304,0: POKE  - 16302,0: POKE  - 1
     6300,0: POKE  - 16297,0: GOTO 1000
3040  GOSUB 3200: PRINT : PRINT D$"BSAVE "FL$",A$
     2000,L$2000": GOTO 3000
3050  GOSUB 3200: PRINT : PRINT D$"BLOAD "FL$",A$
     2000": GOTO 3000
3060  PRINT : PRINT : PRINT "  ERASE PICTURE IN M
     EMORY ?  "BL$;: GET W$: IF W$ <  > "Y" THEN
     3000
3065  CALL 62450: GOTO 3000
3070  HOME : END
3080 F1 = F1 + 1: IF F1 = 2 THEN F1 = 0
3090  VTAB 20: HTAB 2: PRINT "CURSOR IS NOW ";: IF
     F1 THEN  PRINT "OFF ";: GOTO 3100
3095  PRINT "ON ";
3100  GET W$: GOTO 3000
3110  VTAB 20: HTAB 2: PRINT "THE CURSOR IS NOW A
     T (X,Y) "X","Y: VTAB 22: HTAB 2: PRINT "NEW
     (X,Y) ";: INPUT "";X$,Y$
3115  IF X$ = "" OR Y$ = "" THEN 3000
3120 XX =  VAL (X$):YY =  VAL (Y$): IF XX < 0 OR
     XX > 39 OR YY < 0 OR YY > 24 THEN 3110
3130 X =  VAL (X$):Y =  VAL (Y$): GOTO 3000
3200  PRINT : PRINT : INPUT "  FILENAME:";FL$: IF
     FL$ = "" THEN  POP : GOTO 3000
3210  RETURN
```

9.16 SEVENTY CHARACTERS

Since 280 dots are in every line of the high resolution screen, it is possible to draw 70 characters of three dots per side, plus one dot to separate them.

Program SEVENTY CHARACTERS is very similar to FORTY CHARACTERS; the shape table is called ALPHABET 70, and a few values have to be changed to compensate for the different size characters and the different number of characters per line.

Only upper and lower case letters are available in the shape table but you can add extra characters with the shape generator.

```
1  REM  ** SEVENTY CHARACTERS **
10  HIMEM: 25000:D$ =  CHR$ (4): PRINT D$"BLOAD A
     LPHABET 70": POKE 232,168: POKE 233,97: POKE
     230,32: HCOLOR= 3
20  ROT= 0: SCALE= 1:BL$ =  CHR$ (7):FL = 1:BL =
      - 16336: HOME
25  VTAB 6: PRINT "1-RETURN TO HIGH-RES": PRINT :
      PRINT "2-SAVE TO DISK": PRINT : PRINT "3-LO
     AD FROM DISK": PRINT : PRINT "4-CLEAR SCREEN
     "
27  PRINT : PRINT "5-TOGGLE CURSOR": PRINT : PRINT
     "6-MOVE CURSOR": PRINT : PRINT "7-QUIT ";
30  GOTO 3000
1000  GOSUB 2000: GET W$: GOSUB 2010: IF W$ =  CHR$
     (32) OR W$ =  CHR$ (21) THEN  GOSUB 2300: GOTO
     1130
1005  IF W$ =  CHR$ (13) THEN 1140
1006  IF W$ <  >  CHR$ (10) THEN 1010
1007  IF Y >  = 23 THEN  PRINT BL$;: GOTO 1000
1008 Y = Y + 1: GOTO 1000
1010  IF W$ = "" THEN  PRINT BL$;: GOTO 1000
1011  IF W$ <  >  CHR$ (8) THEN 1020
1012  IF X > 0 THEN X = X - 1: GOSUB 2020: GOTO 1
     000
1013  IF Y = 0 THEN  PRINT BL$;: GOTO 1000
1014 Y = Y - 1:X = 69: GOSUB 2020: GOTO 1000
1020  IF W$ <  >  CHR$ (1) THEN 1042
1030 FL = FL + 1: IF FL = 3 THEN FL = 1
1040  GOTO 1000
1042  IF W$ <  >  CHR$ (94) THEN 1047
1043  IF Y = 0 THEN  PRINT BL$;: GOTO 1000
1044 Y = Y - 1: GOTO 1000
1047  IF W$ =  CHR$ (27) THEN 3000
1050 S =  ASC (W$) - 31: ON FL GOTO 1090,1060
1060  IF S >  = 34 AND S <  = 59 THEN  DRAW S + 3
     2 AT X * 4,Y * 8: GOTO 1130
1070  IF S > 1 AND S < 64 THEN  DRAW S AT X * 4,Y
      * 8: GOTO 1130
1080  PRINT BL$;: GOTO 1000
1090  IF S >  = 1 AND S <  = 63 THEN  DRAW S AT X
      * 4,Y * 8: GOTO 1130
1100  PRINT BL$;: GOTO 1000
1130  IF X < 69 THEN X = X + 1: GOTO 1000
1140  IF Y >  = 23 THEN  PRINT BL$;: GOTO 1000
1145 X = 0:Y = Y + 1: GOTO 1170
1150  PRINT BL$;: GOTO 1000
1170  GOTO 1000
2000  IF F1 THEN  RETURN
2002  IF X = 69 AND Y = 23 THEN PX = 273: RETURN
2005 PX = X * 4:PY = Y: DRAW 95 AT PX,Y * 8 + 7 -
      (FL - 1) * 3: RETURN
2010  IF F1 THEN  RETURN
```

```
2012  IF X = 69 AND Y = 23 THEN  RETURN
2015  HCOLOR= 0: DRAW 95 AT PX,PY * 8 + 7 - (FL -
     1) * 3: HCOLOR= 3: RETURN
2020  IF PX > 0 THEN NX = PX:NY = PY: GOTO 2220
2030  IF PY = 0 THEN  PRINT BL$;: POP : GOTO 1000

2040 NY = PY - 1:NX = 280
2220  HCOLOR= 0: DRAW 96 AT NX - 4,NY * 8: HCOLOR=
     3: RETURN
2300  HCOLOR= 0: DRAW 96 AT X * 4,Y * 8 + 1: HCOLOR=
     3: RETURN
3000  POKE  - 16303,0: VTAB 19: HTAB 10: CALL  -
     958
3010  GET W$: IF W$ < "1" OR W$ > "7" THEN 3010
3020  ON  VAL (W$) GOTO 3030,3040,3050,3060,3080,
     3110,3070
3030  POKE  - 16304,0: POKE  - 16302,0: POKE  - 1
     6300,0: POKE  - 16297,0: GOTO 1000
3040  GOSUB 3200: PRINT : PRINT D$"BSAVE "FL$",A$
     2000,L$2000": GOTO 3000
3050  GOSUB 3200: PRINT : PRINT D$"BLOAD "FL$",A$
     2000": GOTO 3000
3060  PRINT : PRINT : PRINT "  ERASE PICTURE IN M
     EMORY ?  "BL$;: GET W$: IF W$ <  > "Y" THEN
     3000
3065  CALL 62450: GOTO 3000
3070  HOME : END
3080 F1 = F1 + 1: IF F1 = 2 THEN F1 = 0
3090  VTAB 20: HTAB 2: PRINT "CURSOR IS NOW ";: IF
     F1 THEN  PRINT "OFF ";: GOTO 3100
3095  PRINT "ON ";
3100  GET W$: GOTO 3000
3110  VTAB 20: HTAB 2: PRINT "THE CURSOR IS NOW A
     T (X,Y) "X","Y: VTAB 22: HTAB 2: PRINT "NEW
      (X,Y) ";: INPUT "";X$,Y$
3115  IF X$ = "" OR Y$ = "" THEN 3000
3120 XX =  VAL (X$):YY =  VAL (Y$): IF XX < 0 OR
     XX > 39 OR YY < 0 OR YY > 24 THEN 3110
3130 X =  VAL (X$):Y =  VAL (Y$): GOTO 3000
3200  PRINT : PRINT : INPUT "  FILENAME:";FL$: IF
     FL$ = "" THEN  POP : GOTO 3000
3210  RETURN
```

9.17 INVERSE

No provision is made to draw characters in INVERSE mode in the two previous programs; yet, using the block (shape 118 in the 40 characters and 96 in the 70 characters alphabets), it becomes an easy task. Programs INVERSE FORTY and INVERSE SEVENTY give you a very simple implementation: the block is drawn in white, and the letter in black.

```
1  REM  ** INVERSE FORTY **
10  HIMEM: 25000: PRINT  CHR$ (4)"BLOAD ALPHABET"
    : HGR : ROT= 0: SCALE= 1: HOME
20  FOR I = 1 TO 40: HCOLOR= 3: DRAW 118 AT (I -
    1) * 7,Y * 8 + 1: HCOLOR= 0: DRAW I AT (I -
    1) * 7,Y * 8: NEXT
25 Y = Y + 1
30  FOR I = 41 TO 80: HCOLOR= 3: DRAW 118 AT (I -
    41) * 7,Y * 8 + 1: HCOLOR= 0: DRAW I AT (I -
    41) * 7,Y * 8: NEXT
35 Y = Y + 1
40  FOR I = 81 TO 118: HCOLOR= 3: DRAW 118 AT (I -
    81) * 7,Y * 8 + 1: HCOLOR= 0: DRAW I AT (I -
    81) * 7,Y * 8: NEXT
45  HCOLOR= 3: DRAW 118 AT (I - 81) * 7,Y * 8 + 1
```

```
1  REM  ** INVERSE SEVENTY **
10  HIMEM: 25000: PRINT  CHR$ (4)"BLOAD ALPHABET
    70": HGR : ROT= 0: SCALE= 1: HOME
20  FOR I = 1 TO 70: HCOLOR= 3: DRAW 96 AT (I - 1
    ) * 4,Y * 8: HCOLOR= 0: DRAW I AT (I - 1) *
    4,Y * 8: NEXT
25 Y = Y + 1
30  FOR I = 71 TO 96: HCOLOR= 3: DRAW 96 AT (I -
    71) * 4,Y * 8: HCOLOR= 0: DRAW I AT (I - 71)
     * 4,Y * 8: NEXT
35  HCOLOR= 3: DRAW 96 AT (I - 71) * 4,Y * 8
```

9.18 EMULATING THE PRINT COMMAND IN HIGH RESOLUTION

Both programs FORTY and SEVENTY let you do the input with the keyboard and respond the way the TEXT mode would, but there is no way to execute an action equivalent to PRINT X$ from within a program.

Programs EMULATOR FORTY and EMULATOR SEVENTY print in the high resolution screen the characters sent in the string variable X$, acting exactly the same way the PRINT command does, except for the scroll, which is done by clearing the screen and starting over again at the top of the screen.

```
1  REM  ** EMULATOR FORTY **
5  LOMEM: 17000
10 D$ =  CHR$ (4): HIMEM: 25000: PRINT D$"BLOAD A
     LPHABET": POKE 232,168: POKE 233,97:LF$ =  CHR$
     (13)
15  HGR : POKE  - 16302,0: HCOLOR= 3: ROT= 0: SCALE=
     1
16  HOME
17 X$ = "APPLE " + "]" +  CHR$ (91) +  CHR$ (13):
     Y = 0:X = 16: GOSUB 10000
20 X = 0:Y = 10:X$ = "PRINT YOUR NAME : ": GOSUB
     10000
30  GET X$: IF X$ =  CHR$ (13) THEN 50
35 Y$ = Y$ + X$
40  GOSUB 10000: GOTO 30
50 X$ =  CHR$ (13) +  CHR$ (13): GOSUB 10000
55 X$ = LF$ + "NOW IN LOWER CASE:" + LF$: GOSUB 1
     0000
60  FOR I = 1 TO  LEN (Y$):W$ =  MID$ (Y$,I,1): IF
     W$ > = "A" AND W$ <  = "Z" THEN T$ = T$ +  CHR$
     ( ASC (W$) + 32): GOTO 90
70 T$ = T$ + W$
90  NEXT :X$ = T$: GOSUB 10000
92 X$ = LF$ + "THE ASCII CODE : " + LF$: GOSUB 10
     000
95  FOR II = 32 TO 170:X$ =  CHR$ (II) + "-" +  STR$
     (II) +  CHR$ (13):X = 17: GOSUB 10000: NEXT

100  FOR II = 1 TO 100:X = 10:Y = 10:X$ =  STR$ (
      II): GOSUB 10000: NEXT
110 X$ =  CHR$ (13) + "]": GOSUB 10000: END
10000  IF X$ = "" THEN  RETURN
10010  FOR I = 1 TO  LEN (X$):W$ =  MID$ (X$,I,1)
     : IF W$ <  > CHR$ (8) THEN 10060
10020  IF X > 0 THEN X = X - 1: GOTO 10050
10030  IF Y = 0 THEN 10050
10040 X = 39:Y = Y - 1
10050  HCOLOR= 0: DRAW 118 AT X * 7,Y * 8: HCOLOR=
     3: GOTO 10180
10060  IF W$ <  > CHR$ (13) THEN 10090
10070  IF Y = 23 THEN 10160
10080 X = 0:Y = Y + 1: GOTO 10180
10090  IF W$ <  > CHR$ (10) THEN 10120
10100  IF Y = 23 THEN
10110 Y = Y + 1: GOTO 10180
10120  IF W$ =  CHR$ (32) THEN  HCOLOR= 0: DRAW 1
     18 AT X * 7,Y * 8: HCOLOR= 3: GOTO 10150
10130  IF W$ < " " OR W$ >  CHR$ (150) THEN 10150

10140  HCOLOR= 0: DRAW 118 AT X * 7,Y * 8 + 1: HCOLOR=
     3: DRAW  ASC (W$) - 32 AT X * 7,Y * 8
10150 X = X + 1: IF X < 39 THEN 10180
```

```
10160  IF Y = 23 THEN  HGR : POKE  - 16302,0:X =
     0:Y = 0: GOTO 10180
10170 Y = Y + 1:X = 0
10180  NEXT
10190  RETURN


1  REM  ** EMULATOR SEVENTY **
5  LOMEM: 17000
10 D$ =  CHR$ (4): HIMEM: 25000: PRINT D$"BLOAD A
     LPHABET 70": POKE 232,168: POKE 233,97:LF$ =
      CHR$ (13)
15  HGR : POKE  - 16302,0: HCOLOR= 3: ROT= 0: SCALE=
     1
16  HOME
17 X$ = "APPLE " + "]" +  CHR$ (91) +  CHR$ (13):
     Y = 0:X = 31: GOSUB 10000
20 X = 0:Y = 10:X$ = "PRINT YOUR NAME : ": GOSUB
     10000
30  GET X$: IF X$ =  CHR$ (13) THEN 50
35 Y$ = Y$ + X$
40  GOSUB 10000: GOTO 30
50 X$ =  CHR$ (13) +  CHR$ (13): GOSUB 10000
55 X$ = LF$ + "NOW IN LOWER CASE:" + LF$: GOSUB 1
     0000
60  FOR I = 1 TO  LEN (Y$):W$ =  MID$ (Y$,I,1): IF
     W$ >  = "A" AND W$ <  = "Z" THEN T$ = T$ +  CHR$
     ( ASC (W$) + 32): GOTO 90
70 T$ = T$ + W$
90  NEXT :X$ = T$: GOSUB 10000
92 X$ = LF$ + "THE ASCII CODE : " + LF$: GOSUB 10
     000
95  FOR II = 32 TO 140:X$ =  CHR$ (II) + "-" +  STR$
     (II) +  CHR$ (13):X = 30: GOSUB 10000: NEXT

100  FOR II = 1 TO 100:X = 10:Y = 10:X$ =  STR$ (
     II): GOSUB 10000: NEXT
110 X$ =  CHR$ (13) + "]": GOSUB 10000: END
10000  IF X$ = "" THEN  RETURN
10010  FOR I = 1 TO  LEN (X$):W$ =  MID$ (X$,I,1)
     : IF W$ <  >  CHR$ (8) THEN 10060
10020  IF X > 0 THEN X = X - 1: GOTO 10050
10030  IF Y = 0 THEN 10050
10040 X = 69:Y = Y - 1
10050  HCOLOR= 0: DRAW 96 AT X * 4,Y * 8: HCOLOR=
     3: GOTO 10180
10060  IF W$ <  >  CHR$ (13) THEN 10090
10070  IF Y = 23 THEN 10160
10080 X = 0:Y = Y + 1: GOTO 10180
10090  IF W$ <  >  CHR$ (10) THEN 10120
10100  IF Y = 23 THEN
```

```
10110 Y = Y + 1: GOTO 10180
10120  IF W$ =  CHR$ (32) THEN  HCOLOR= 0: DRAW 9
     6 AT X * 4,Y * 8: HCOLOR= 3: GOTO 10150
10130  IF W$ < " " OR W$ >  CHR$ (127) THEN 10150

10140  HCOLOR= 0: DRAW 96 AT X * 4,Y * 8: HCOLOR=
     3: DRAW  ASC (W$) - 31 AT X * 4,Y * 8
10150 X = X + 1: IF X < 69 THEN 10180
10160  IF Y = 23 THEN  HGR : POKE  - 16302,0:X =
     0:Y = 0: GOTO 10180
10170 Y = Y + 1:X = 0
10180  NEXT
10190  RETURN
```

9.19 SEVENTY BY THIRTY-TWO CHARACTERS

A character made in a matrix 3 by 5 dots is still very readable in a black and white or high resolution color monitor. If you allow one dot separation horizontally and vertically, 32 lines of 70 characters can be used.

9.20 GENERATING THE ALPHABETIC SHAPE TABLES

The following two programs produce the shape tables needed for all the programs that need characters while in high resolution graphics in this chapter. If you bought the disk with the programs of this book, you do not need to type them, as the shape tables are already available under the names ALPHABET and ALPHABET 70.

To SAVE the shape table to disk, type

```
BSAVE ALPHABET, A30000, L1470
```

or

```
BSAVE ALPHABET 70, A30000, L927
```

depending on which alphabet creator you ran. The 30000 is the address you specified at the beginning of the program, and can be replaced by any other legal memory address.

```
1  REM  ** ALPHABET CREATOR **
10  HOME : INPUT "STARTING ADDRESS:";SA
20  PRINT : INPUT "SET HIMEM ? ";W$: IF  LEFT$ ("
     YES", LEN (W$)) = W$ THEN  HIMEM: SA
30  READ N
40  FOR I = 0 TO N - 1: READ W: POKE SA + I,W: NEXT
50 S =  INT (SA / 256): POKE 233,S
60 W = SA - 256 * S: POKE 232,W
1000  DATA 1470,118,0,238,0,244,0,249,0,8,1,21,1,
     32,1,45,1,49,1,57,1,65,1,79,1,88,1,93,1,98,1
     ,103,1,111,1,126,1,134,1,144,1,154,1,167,1,1
     80,1,194,1,202,1,215,1,229,1,233,1,238,1,247
     ,1,255,1,7,2,15,2,30,2,44,2,57,2,67,2,78,2,9
     2,2,104
1010  DATA 2,116,2,129,2,137,2,146,2,161,2,168,2,
     183,2,196,2,207,2,217,2,229,2,243,2,255,2,7,
     3,17,3,27,3,40,3,52,3,62,3,72,3,88,3,94,3,10
     9,3,116,3,122,3,126,3,137,3,147,3,156,3,168,
     3,179,3,189,3,203,3,213,3,221,3,230,3,242,3,
     249,3,6
1020  DATA 4,17,4,27,4,37,4,49,4,59,4,71,4,81,4,9
     0,4,99,4,110,4,121,4,133,4,144,4,151,4,157,4
     ,163,4,169,4,180,4,191,4,201,4,211,4,219,4,2
     28,4,241,4,255,4,10,5,24,5,40,5,51,5,66,5,78
     ,5,87,5,97,5,107,5,117,5,129,5,137,5,147,5,1
     56,5,160
1030  DATA 5,9,54,54,22,4,0,49,110,36,6,0,49,62,5
     3,62,53,110,36,47,61,36,47,61,36,6,0,9,46,61
     ,63,23,14,45,21,30,55,60,47,0,46,108,17,30,3
     0,30,30,77,46,52,0,113,31,118,197,177,14,12,
     22,31,63,32,4,0,9,54,4,0,9,30,30,54,14,14,4,
     0,9,14
1040  DATA 14,54,30,30,4,0,114,37,52,46,197,177,2
     7,62,23,141,36,173,4,0,137,54,63,45,45,63,54
     ,4,0,137,146,246,5,0,146,45,45,7,0,137,146,1
     8,4,0,73,17,30,30,30,30,4,0,41,173,54,54,30,
     63,199,33,36,108,10,30,23,5,0,9,62,53,54,54,
     47,61,0
1050  DATA 42,40,173,246,191,30,46,45,61,0,45,45,
     246,30,173,246,63,199,41,0,73,54,54,61,63,39
     ,12,12,53,54,54,4,0,45,45,63,63,54,45,173,54
     ,30,63,199,41,0,9,45,63,23,23,46,45,21,246,6
     3,199,33,4,0,45,45,246,30,30,54,4,0,41,173,2
     46,63
1060  DATA 199,33,182,118,45,197,33,4,0,41,173,54
     ,63,63,32,77,49,54,30,30,63,5,0,137,178,4,0,
     137,178,246,5,0,73,30,30,30,14,14,14,4,0,18,
     45,45,22,63,63,5,0,113,14,14,30,30,30,4,0,42
     ,40,173,30,30,182,4,0,50,54,118,45,37,192,57
     ,39,54
1070  DATA 37,37,228,63,7,0,9,30,30,54,54,36,45,4
     5,54,36,36,28,6,0,45,173,246,63,39,52,54,54,
      45,45,32,6,0,41,173,223,51,54,118,45,197,33,
      0,45,173,54,54,30,63,39,36,36,4,0,45,45,63,6
      3,54,46,45,63,55,54,45,45,7,0,45,45,63,63,54
      ,46,45
1080   DATA 63,55,54,4,0,41,45,63,191,54,54,14,45,
      37,60,5,0,54,54,54,36,44,45,37,36,54,54,54,4
      ,0,41,61,54,54,54,47,61,0,73,49,54,54,30,63,
      199,41,0,54,54,54,36,44,21,21,21,196,193,217
      ,40,40,48,0,54,54,54,45,45,7,0,54,54,54,36,3
      6,44,21
1090   DATA 38,12,37,54,54,54,6,0,54,54,54,36,36,1
      73,21,53,38,36,36,52,0,41,173,54,54,30,63,19
      9,33,36,52,0,45,173,246,63,39,52,54,54,4,0,4
      1,173,54,182,199,57,176,63,32,36,52,0,45,173
      ,246,63,39,52,54,54,36,13,14,14,4,0,41,173,2
      23,51,14
1100   DATA 45,21,246,63,199,41,0,45,45,63,54,54,5
      4,4,0,54,54,118,45,197,33,36,36,6,0,54,54,14
      ,14,197,41,32,36,52,0,54,54,54,44,197,33,118
      ,53,36,36,36,6,0,118,14,30,30,110,9,228,196,
      41,32,6,0,118,14,12,12,52,150,27,54,4,0,45,4
      5,246,30
1110   DATA 30,30,46,45,61,0,45,45,63,63,54,54,54,
      37,36,36,54,54,46,45,7,0,114,14,14,14,4,0,45
      ,45,54,54,54,39,36,36,52,54,54,62,63,7,0,146
      ,42,40,168,21,7,0,146,146,45,45,7,0,169,21,6
      ,0,82,45,21,62,63,23,14,45,37,6,0,54,54,54,4
      5,45,32
1120   DATA 228,63,5,0,82,45,21,223,51,118,45,61,0
      ,73,49,62,63,23,54,14,45,37,36,4,0,82,45,21,
      62,63,39,54,14,45,61,0,9,173,223,54,47,45,63
      ,54,38,0,82,45,21,54,63,63,32,118,45,53,30,6
      3,5,0,54,54,54,36,36,45,173,54,38,0,9,22,47,
      54,54,47
1130   DATA 61,0,73,22,47,54,54,30,63,40,0,54,54,5
      4,36,45,197,41,176,26,21,4,0,41,54,54,54,47,
      61,0,18,54,54,36,36,173,54,36,12,53,54,38,0,
      18,54,54,36,44,197,41,21,54,54,0,82,45,21,54
      ,30,63,199,33,52,0,18,45,173,246,63,39,52,54
      ,38,0,82
1140   DATA 45,53,54,63,63,32,118,45,53,38,0,18,46
      ,197,41,61,247,55,54,4,0,82,45,61,63,23,14,4
      5,21,30,63,47,0,49,62,45,61,55,54,14,45,56,0
      ,18,54,118,109,60,37,36,6,0,18,54,14,14,197,
      41,32,52,0,18,54,54,45,32,52,118,37,36,52,0,
      18,14,14
1150   DATA 30,30,77,225,196,41,48,0,18,54,14,45,3
      7,36,54,54,30,63,5,0,18,45,45,30,30,30,30,45
      ,45,7,0,73,30,246,21,118,4,0,9,54,22,54,4,0,
      169,118,30,246,4,0,42,40,46,197,49,0,18,37,3
      7,54,37,46,63,54,54,4,0,9,54,54,63,53,45,44,
      63,54,4
```

```
1160  DATA 0,74,30,30,45,45,63,55,21,5,0,137,21,4
     6,63,63,45,53,30,5,0,169,21,46,63,183,45,61,
     0,73,30,30,62,45,181,63,47,0,54,22,197,41,40
     ,40,176,58,53,62,54,61,0,105,46,62,60,63,46,
     53,47,53,62,60,46,38,0,137,62,62,53,53,44,60
     ,45,39,6
1170  DATA 0,41,53,63,53,62,55,45,45,60,55,54,47,
     61,0,9,62,46,44,54,53,63,60,55,53,45,55,62,4
     5,7,0,82,173,197,177,51,21,31,63,32,52,0,10,
     45,21,30,63,39,54,46,45,197,185,63,55,38,0,4
     5,45,254,115,14,30,30,30,45,45,60,0,82,45,61
     ,54,246
1180  DATA 63,32,36,0,45,45,55,54,54,254,36,36,52
     ,0,45,45,38,63,63,54,54,54,4,0,9,45,63,54,54
     ,54,60,199,57,0,41,181,21,54,254,27,36,36,46
     ,197,41,0,9,54,54,62,55,45,4,0,9,46,21,31,54
     ,62,55,45,4,0,73,21,31,54,54,246,199,41,0,45
     ,45,7,0
1190  DATA 44,45,45,53,63,63,63,46,45,45,53,63,63
     ,63,46,45,45,53,63,63,63,46,45,45,53,63,63,6
     3,5,0
```

```
1  REM  ** ALPHABET 70 CREATOR **
10  HOME : INPUT "STARTING ADDRESS:";SA
20  PRINT : INPUT "SET HIMEM ? ";W$: IF  LEFT$ ("
     YES", LEN (W$)) = W$ THEN  HIMEM: SA
30  READ N
40  FOR I = 0 TO N - 1: READ W: POKE SA + I,W: NEXT

50 S =  INT (SA / 256): POKE 233,S
60 W = SA - 256 * S: POKE 232,W
1000  DATA 927,96,0,3,0,196,0,201,0,204,0,215,0,2
     24,0,230,0,239,0,242,0,250,0,1,1,11,1,17,1,2
     4,1,28,1,33,1,38,1,47,1,54,1,62,1,70,1,78,1,
     87,1,96,1,102,1,111,1,120,1,124,1,129,1,136,
     1,142,1,148,1,155,1,165,1,176,1,187,1,195,1,
     204,1
1010  DATA 212,1,219,1,228,1,238,1,243,1,250,1,4,
     2,10,2,20,2,29,2,38,2,48,2,58,2,68,2,76,2,82
     ,2,91,2,100,2,110,2,118,2,126,2,134,2,141,2,
     146,2,153,2,157,2,162,2,165,2,173,2,181,2,18
     9,2,198,2,208,2,215,2,226,2,234,2,239,2,246,
     2,0,3,7
1020  DATA 3,17,3,25,3,33,3,42,3,52,3,60,3,69,3,7
     5,3,83,3,91,3,100,3,108,3,116,3,123,3,130,3,
     135,3,141,3,144,3,142,3,49,54,54,58,0,110,60
     ,0,54,54,54,13,36,36,36,151,22,6,0,169,63,54
     ,45,54,63,14,4,0,106,186,23,141,7,0,41,62,46
     ,222,54
1030  DATA 14,37,36,0,49,6,0,9,30,30,54,14,14,4,0
```

```
      ,14,14,54,30,30,4,0,178,22,13,28,36,12,22,22
      ,7,0,82,21,63,14,5,0,146,41,62,14,30,4,0,146
      ,45,6,0,146,82,53,39,0,82,241,30,6,0,50,54,5
      4,41,32,36,36,51,0,241,53,54,54,111,4,0,42,1
      68,246
1040   DATA 30,54,45,4,0,42,168,246,14,246,28,6,0,
      54,46,37,164,182,54,7,0,45,222,46,21,54,30,2
      8,6,0,169,31,54,54,14,12,36,63,0,45,54,23,54
      ,46,0,50,22,118,12,228,12,228,6,0,50,14,37,2
      28,142,54,54,7,0,82,22,6,0,82,22,246,7,0,137
      ,23,23
1050   DATA 21,21,7,0,18,45,22,63,4,0,170,21,23,23
      ,5,0,46,40,54,62,182,7,0,42,168,54,54,30,28,
      36,53,6,0,50,54,54,78,32,60,12,36,28,6,0,54,
      54,54,14,40,32,28,12,228,6,0,45,254,54,54,46
      ,37,7,0,54,54,54,45,32,36,36,59,0,45,222,54,
      245,54
1060   DATA 45,5,0,45,222,54,245,54,5,0,45,254,54,
      45,54,62,39,36,0,54,54,54,13,36,60,44,36,7,0
      ,49,54,54,54,0,41,54,54,54,63,4,0,54,54,54,1
      3,36,39,44,36,7,0,54,54,54,45,4,0,54,54,54,1
      3,36,36,36,23,6,0,54,54,54,13,36,36,60,7,0,5
      4,54,54
1070   DATA 45,36,36,36,63,0,54,54,54,13,24,40,36,
      36,63,0,50,54,53,46,4,32,36,39,7,0,54,54,54,
      13,36,35,37,36,63,0,169,31,54,113,246,28,6,0
      ,45,30,54,54,54,0,54,54,54,45,36,36,36,7,0,5
      4,54,46,46,32,36,36,7,0,54,54,54,13,60,12,36
      ,36,7,0
1080   DATA 54,146,110,228,36,12,60,0,54,150,10,36
      ,36,37,60,0,45,54,30,30,54,45,4,0,45,222,54,
      54,46,37,0,18,21,21,4,0,45,54,54,54,63,4,0,9
      8,14,4,0,146,146,45,4,0,49,6,0,18,45,54,63,5
      4,45,36,0,54,54,54,45,36,36,63,0,18,45,63,54
      ,54,45,4
1090   DATA 0,9,54,54,54,63,36,36,45,0,18,45,54,63
      ,36,54,54,45,4,0,113,31,54,46,55,46,0,18,45,
      54,63,36,45,54,54,63,4,0,54,54,54,13,36,36,6
      3,0,177,54,54,7,0,9,22,54,54,63,4,0,54,54,54
      ,36,173,38,196,33,7,0,53,54,54,62,45,7,0,18,
      46,53,63
1100   DATA 46,53,60,55,4,0,18,46,53,54,31,36,4,0,
      18,45,54,54,63,36,36,0,18,45,54,63,36,54,54,
      5,0,18,45,54,63,36,45,54,54,7,0,18,46,197,24
      1,55,54,5,0,18,45,63,54,45,54,63,4,0,54,61,5
      4,118,4,0,18,54,54,45,36,36,4,0,18,54,118,19
      7,33,36
1110   DATA 4,0,18,54,54,44,53,36,36,4,0,18,182,11
      0,228,197,33,7,0,18,54,45,36,54,54,63,0,18,4
      5,246,23,46,45,0,9,30,246,21,118,4,0,49,182,
      54,4,0,21,118,23,246,4,0,45,7,0,45,53,63,55,
      45,53,63,55,45,53,63,55,45,61,0
```

10

SOUND

10.1 PRODUCING SOUNDS

There are three ways to produce a sound in the Apple: the bell character (CHR$ (7) or ctrl-G), and PEEKing into locations –16336 and –16352, which causes the speaker or the cassette output to emit a very short click.

Since the only variation with the bell character is repetition, it does not provide a very interesting range of possibilities.

The click, being so short, allows more varied combinations with Applesoft.

If many clicks are produced in a very fast succession, they combine to make a noise or a musical note. Try the following:

```
10 FOR I = 1 TO 100 : X = PEEK (–16336) : NEXT
```

This produces a somewhat musical note. Now try:

```
10 D = –16336
20 FOR I = 1 TO 100 : X = PEEK (D) : NEXT
```

As you can see, the second version has a higher pitch. It takes Applesoft a long time to convert a number into its floating point equivalent, whereas if it has the number in a variable, it can use it directly, without conversion.

To make the pitch change, different lapses of silence can be added between clicks.

Program BURP DOWN has a big loop in line 300, in which the variable I is incremented from 1 to 40, and a click is emitted. Line 310 takes a little time (as I increases, the J loop becomes longer;

therefore, the distance between clicks increases). The result is (as the name says) a descending sound.

```
1   REM   ** BURP DOWN **
10 D =   - 16336
300   FOR I = 1 TO 40:X =   PEEK (D)
310   FOR J = 1 TO I: NEXT
320   NEXT I
```

Program BURP UP begins with a low pitch and ascends.

To make the click a little louder, the X = PEEK (D) of line 300 has been changed to X = PEEK (D) + PEEK (D) in line 400 of program BURP DOWN.

```
1   REM   ** BURP UP **
10 D =   - 16336
400   FOR I = 35 TO 1 STEP   - 1:X =   PEEK (D) +   PEEK
      (D)
410   FOR J = 1 TO I: NEXT
420   NEXT I
```

Noise

Program NOISE has lapses of silence of random duration; the resulting sound is a noise somewhat like a geiger counter.

```
1   REM   ** NOISE **
10 D =   - 16336
1000   FOR I = 1 TO 500
1010 X =   PEEK (D): FOR J = 1 TO   RND (1) * 30: NEXT

1020   NEXT
```

Program Rattle

If several clicks are produced before the silence lapse, the sound produced resembles a rattle.

```
1   REM   ** RATTLE **
600   FOR I = 1 TO 20
610   FOR J = 1 TO 5:X =   PEEK ( - 16336): NEXT
620   FOR J = 1 TO 15: NEXT
630   NEXT
```

10.2 AMPLIFYING THE SOUNDS

If you want to hear the music produced by the computer through your sound system, a cable can be connected to the OUTPUT jack at the back of the Apple and the sounds amplified. The location where the PEEK produces the click must be changed for the one that goes to the cassette output. Therefore, every −16336 has to be changed to −16352.

Siren

Before going into more applications of the scale, let me introduce a different sound, because we will be dealing with the scale for quite a while. Add program SIREN to your collection of useful sound routines.

```
1  REM  ** SIREN **
210  FOR I = 1 TO 19: READ W: POKE I + 767,W: NEXT

220  POKE 6,100: CALL 768
10010  DATA 173,48,192,136,208,4,198,7,240,8,202,
     208,246,166,7,76,0,3,96
```

10.3 PRODUCING MUSICAL NOTES

Unfortunately, not very many more sounds can be produced with Applesoft alone. We will have to resort to a very short machine language routine which will let us produce much more interesting noises and music.

Program MUSIC POKEs this machine language routine between locations 770 and 790.

The values prompted for duration and pitch are POKED into locations 768 and 769, and the routine is executed with a CALL 770. Since the values are going to be POKEd into one memory location, the smallest number accepted will be 0 and the highest 255. Those are the limits of the routine, and the notes produced do not correspond to the standard musical scale.

To change the sound output from the speaker to the cassette output so that the music can be amplified, change the 48 (the second number in the DATA list) to 32.

```
1  REM  ** MUSIC **
10  FOR I = 770 TO 790: READ J: POKE I,J: NEXT
15  HOME
20  VTAB 6: INPUT "PITCH:";P
30  PRINT : INPUT "DURATION:";D
40  POKE 768,P: POKE 769,D: CALL 770: GOTO 15
10000  DATA 173,48,192,136,208,5,206,1,3,240,9,20
     2,208,245,174,0,3,76,2,3,96
```

Cascade

Program CASCADE POKEs ascending values into the pitch location, producing exactly what the name says—a cascade. Based on this one, it is trivial to do the program CASCADE DOWN.

```
1  REM  ** CASCADE **
700  FOR I = 770 TO 790: READ J: POKE I,J: NEXT
710  FOR I = 5 TO 150: POKE 768,I: POKE 769,7
720  CALL 770: NEXT
10000  DATA 173,48,192,136,208,5,206,1,3,240,9,20
     2,208,245,174,0,3,76,2,3,96
```

Computer Music

When the pitch and duration are random numbers, the sound produced is what most people would call "computer music." Program COMPUTER SONG plays a nice computeresque song.

```
1  REM  ** COMPUTER SONG **
100  FOR I = 770 TO 790: READ J: POKE I,J: NEXT
110  FOR I = 1 TO 100: POKE 768, RND (3) * 255: POKE
     769, RND (3) * 10 + 1
120  CALL 770: NEXT
10000  DATA 173,48,192,136,208,5,206,1,3,240,9,20
     2,208,245,174,0,3,76,2,3,96
```

10.4 THE MUSICAL SCALE

To produce notes that correspond to the Bach scale, the one used in most western music (8 tones and 12 semi-tones), specific values must be found that match the true pitches as closely as possible. Since the values POKEd are integers which do not follow the frequency relations found in music, some of the notes sound slightly off-key, especially those in the high part of the scale, where the notes are closer together. Besides, the notes do not correspond exactly to the real pitches, but since they have the correct relation among themselves, the resulting sound is perfectly musical. In other words, you might not be able to play along with an orchestra.

Program notes include as data the values that correspond to the musical scale, increasing by semi-tones. The whole range of 49 notes reaches four octaves and one semi-tone.

```
1   REM   ** NOTES **
10   FOR I = 770 TO 790: READ J: POKE I,J: NEXT
30   READ N: DIM N(N)
40   FOR I = 1 TO N: READ N(I): NEXT
50   FOR I = 1 TO N: POKE 768,N(I)
60   POKE 769,90: CALL 770
65   NEXT
1000   DATA  173,48,192,136,208,5,206,1,3,240,9,20
     2,208,245,174,0,3,76,2,3,96
1500   DATA    49
2000   DATA  255,240,228,215,204,193,185,171,162,1
     53,145,136,128
2100   DATA 120,114,107,102,96,92,85,81,76,72,68,6
     4
2200   DATA 60,56,53,50,48,45,42,40,37,35,33,31
2300   DATA 29,28,25,24,23,21,20,19,18,17,16,15
```

10.5 COMPUTER PIANO

Program PIANO uses the values corresponding to the musical scale to play songs with the keyboard. The range is limited to one octave and four notes, including semi-tones.

Because of this limitation, there is the option of changing the height of the keyboard. The number entered is taken as the lowest note in the piano. Therefore, as higher heights are chosen, the entire keyboard is shifted. This does not change the musical effect, since all the notes keep the same frequency relation among themselves.

The keys corresponding to the notes are given in Table 10.1.

Table 10.1. Equivalences between keys and notes. (The number after the note indicates the octave)

KEY	NOTE
A	A1
W	A1 # or B1 b
S	B1
D	C2
R	C2 # or D2 b
F	D2
T	D2 # or E2 b
G	E2
H	F2
U	F2 # or G2 b
J	G2
I	G2 # or A2 b
K	A2
O	A2 # or B2 b
L	B2
;	C3
P	C3 # or D3 b
<-	D3
->	E3

The DATA in line 1000 is the machine language music routine. Line 1500 contains the number of notes available (53), and the rest of the data are the values corresponding to the different notes. In line 10 the music routine is READ and POKEd. Line 30 READs the number of notes and DIMensions an array (N ( )) to store the values of the heights of the notes, which are READ by line 40. The height is prompted for, and not accepted if it is outside the range.

Line 80 shifts all the elements of the array, so that the values correspond to the height desired.

Line 500 is the note-entry loop. It compares W$ with each valid note, and assigns a number to N which corresponds to its relative height in the scale. Line 800 (called by every line) plays the note using N as the index of N ( ). The loop is repeated until a ctrl-C is caught by line 690, and the program starts once more, allowing a change of height.

To go up one octave, 12 has to be added to the previous height (the number of semi-tones in an octave).

```
1   REM   ** PIANO **
10   FOR I = 770 TO 790: READ J: POKE I,J: NEXT
30   READ N: DIM N(N)
40   FOR I = 1 TO N: READ N(I): NEXT
45   HOME : VTAB 10
50   INPUT "HEIGHT:";W$: IF W$ = "" THEN  HOME : END

60 H =  VAL (W$): IF H < 1 OR H > 33 THEN 45
70   HOME : VTAB 10: HTAB 13: INVERSE : PRINT "PIA
     NO WORKING";: NORMAL
80   FOR I = 1 TO N - H:N(I) = N(H + I - 1): NEXT

500   GET W$: IF W$ = "A" THEN N = 1: GOTO 800
510   IF W$ = "W" THEN N = 2: GOTO 800
520   IF W$ = "S" THEN N = 3: GOTO 800
530   IF W$ = "D" THEN N = 4: GOTO 800
540   IF W$ = "R" THEN N = 5: GOTO 800
550   IF W$ = "F" THEN N = 6: GOTO 800
560   IF W$ = "T" THEN N = 7: GOTO 800
570   IF W$ = "G" THEN N = 8: GOTO 800
580   IF W$ = "H" THEN N = 9: GOTO 800
590   IF W$ = "U" THEN N = 10: GOTO 800
600   IF W$ = "J" THEN N = 11: GOTO 800
610   IF W$ = "I" THEN N = 12: GOTO 800
620   IF W$ = "K" THEN N = 13: GOTO 800
630   IF W$ = "O" THEN N = 14: GOTO 800
640   IF W$ = "L" THEN N = 15: GOTO 800
650   IF W$ = ";" THEN N = 16: GOTO 800
660   IF W$ = "P" THEN N = 17: GOTO 800
670   IF W$ =  CHR$ (8) THEN N = 18: GOTO 800
680   IF W$ =  CHR$ (21) THEN N = 20: GOTO 800
690   IF W$ =  CHR$ (3) THEN  RUN
700   GOTO 500
800   POKE 768,N(N): POKE 769,30: CALL 770: GOTO 5
      00
1000   DATA  173,48,192,136,208,5,206,1,3,240,9,20
      2,208,245,174,0,3,76,2,3,96
1500   DATA  53
2000   DATA  255,240,228,215,204,193,185,171,162,1
      53,145,136,128
2100   DATA 120,114,107,102,96,92,85,81,76,72,68,6
      4
2200   DATA 60,56,53,50,48,45,42,40,37,35,33,31
2300   DATA 29,28,25,24,23,21,20,19,18,17,16,15
3400   DATA  14,13,12,11
```

10.6 STORING THE MELODIES

Program PIANO has a limitation (among others): After you play the song, there is no record of it, and if you want to hear the song once more, you must play it again.

Program RECORDER is very similar to program PIANO, but it stores the notes and does several things with them.

The main body is a copy of program PIANO; a menu has been added at the beginning, and the height is prompted for in line 1120.

Subroutine 1100 reads the elements of N ( ) every time, to allow for changes in height without erasing the previous notes.

The options on the menu are:

1-PLAY AND RECORD
2-PLAYBACK
3-STORE THE NOTES
4-READ A SONG
5-CREATE A PROGRAM
6-QUIT

1-PLAY AND RECORD

Line 70 calls subroutine 1100, READs the notes into N ( ), and enters loop 500. Line 800, besides playing the notes, stores the corresponding relative height (N) in an array M% ( ).

L is used as the index, and therefore is incremented every time a new note is stored.

When a ctrl-C is captured in line 690, L is set to L-1, so that when the song is played back, it produces exactly the notes played (L is 1 greater than the last note stored). If this first option is called once more from the menu, the height can be changed, and therefore the pitch can be higher or lower in the same song, allowing the use of the full range of notes.

2-PLAYBACK

Line 900 prompts for a speed. Line 950 plays the notes stored in M% ( ), and line 960 pauses while a loop counts from 1 to D (the speed desired).

3-STORE THE NOTES

Line 4050 prompts for a filename, and as usual, returns to the menu if answered with a <return>. Lines 60 to 80 PRINT L (number of notes) and M% ( ) to the disk.

4-READ A SONG

Line 4100 prompts for the filename, and lines 4110 to 4130 read L and M% ( ) from the disk.

5-CREATE A PROGRAM

Line 5000 OPENs the file DUMMY. Line 5015 PRINTs "NEW", so that when this program is brought into memory, it does not conflict with a program already there. Lines 5020 to 5050 PRINT to the disk the following part of the program:

```
10 DIM M% (500) : FOR I = 770 TO 790 : READ J : POKE I, J : NEXT
20 READ N : FOR I = 1 TO N : READ M% (I) : NEXT
30 HOME : VTAB 10 : INPUT"SPEED : ";W$ : D = VAL (W$)
50 FOR I = 1 TO N : POKE 768, M% (I) : POKE 769, 30 : CALL 770
60 FOR J = 1 TO D : NEXT : NEXT
100 DATA ... (the music routine)
```

Lines 5055 to 5110 create the DATA lines with the values corresponding to M% ( ), and RUN is added to the file. The end of line 5110 EXECs the file DUMMY. Therefore, after choosing this option (CREATE PROGRAM), you are left with the new program in memory.

This way, you can play songs with this program and add them to any other program. To merge the music one with an already existing program, use the RENUMBER program available in the SYSTEM MASTER diskette.

```
1  REM  ** RECORDER **
10 D$ =  CHR$ (4):Q$ =  CHR$ (34): FOR I = 770 TO
     790: READ J: POKE I,J: NEXT
20  READ N: DIM N(N),M%(1000):L = 0
30  GOTO 4000
70  GOSUB 1100
100  HOME : VTAB 10: HTAB 13: INVERSE : PRINT "PI
     AND WORKING";: NORMAL
```

```
500  GET W$: IF W$ = "A" THEN N = 1: GOTO 800
510  IF W$ = "W" THEN N = 2: GOTO 800
520  IF W$ = "S" THEN N = 3: GOTO 800
530  IF W$ = "D" THEN N = 4: GOTO 800
540  IF W$ = "R" THEN N = 5: GOTO 800
550  IF W$ = "F" THEN N = 6: GOTO 800
560  IF W$ = "T" THEN N = 7: GOTO 800
570  IF W$ = "G" THEN N = 8: GOTO 800
580  IF W$ = "H" THEN N = 9: GOTO 800
590  IF W$ = "U" THEN N = 10: GOTO 800
600  IF W$ = "J" THEN N = 11: GOTO 800
610  IF W$ = "I" THEN N = 12: GOTO 800
620  IF W$ = "K" THEN N = 13: GOTO 800
630  IF W$ = "O" THEN N = 14: GOTO 800
640  IF W$ = "L" THEN N = 15: GOTO 800
650  IF W$ = ";" THEN N = 16: GOTO 800
660  IF W$ = "P" THEN N = 17: GOTO 800
670  IF W$ =  CHR$ (8) THEN N = 18: GOTO 800
680  IF W$ =  CHR$ (21) THEN N = 20: GOTO 800
690  IF W$ =  CHR$ (3) THEN L = L - 1: GOTO 4000
700  GOTO 500
800  POKE 768,N(N): POKE 769,30: CALL 770:M%(L) =
     N(N):L = L + 1: GOTO 500
900  HOME : VTAB 10: PRINT "SPEED:";: INPUT "";W$
     :D =  VAL (W$): INVERSE : PRINT : HTAB 17: PRINT
     "PLAYING": NORMAL
950  FOR I = 0 TO L: POKE 768,M%(I): POKE 769,30:
      CALL 770
960  FOR J = 1 TO D: NEXT : NEXT : GOTO 4000
1000  DATA  173,48,192,136,208,5,206,1,3,240,9,20
     2,208,245,174,0,3,76,2,3,96
1100  RESTORE : FOR I = 0 TO 20: READ J: NEXT
1110  READ N: FOR I = 1 TO N: READ N(I): NEXT
1120  HOME : VTAB 10: INPUT "HEIGHT:";W$: IF W$ =
     "" THEN  RETURN
1130 H =  VAL (W$): IF H < 1 OR H > 33 THEN 1120
1140  FOR I = 1 TO N - H:N(I) = N(H + I - 1): NEXT
     : RETURN
1500  DATA  53
2000  DATA  255,240,228,215,204,193,185,171,162,1
     53,145,136,128
2100  DATA 120,114,107,102,96,92,85,81,76,72,68,6
     4
2200  DATA 60,56,53,50,48,45,42,40,37,35,33,31
2300  DATA 29,28,25,24,23,21,20,19,18,17,16,15
3400  DATA  14,13,12,11
4000  HOME : VTAB 10: PRINT "1-PLAY AND RECORD": PRINT
     "2-PLAYBACK": PRINT "3-STORE THE NOTES"
4005  PRINT "4-READ A SONG": PRINT "5-PRODUCE A P
     ROGRAM": PRINT "6-QUIT";
4010  GET W$:V =  VAL (W$): IF V < 1 OR V > 6 THEN
     4010
```

```
4020  ON V GOTO 70,900,4050,4100,5000,6000
4050  PRINT : PRINT "  FILENAME:";: INPUT "";FL$:
      IF FL$ = "" THEN 4050
4060  PRINT  CHR$ (4)"OPEN "FL$: PRINT  CHR$ (4)"
     WRITE "FL$
4070  PRINT L: FOR I = 0 TO L: PRINT M%(I): NEXT

4080  PRINT  CHR$ (4)"CLOSE": GOTO 4000
4100  PRINT : PRINT "  FILENAME:";: INPUT "";FL$:
      IF FL$ = "" THEN 4100
4110  PRINT D$"OPEN "FL$: PRINT D$"READ "FL$
4120  INPUT L: FOR I = 0 TO L: INPUT M%(I): NEXT

4130  PRINT D$"CLOSE": GOTO 4000
5000  PRINT : PRINT "FILENAME:";: INPUT "";FL$
5010  RESTORE : PRINT D$"OPEN "FL$: PRINT D$"WRIT
     E "FL$
5015  PRINT "NEW"
5020  PRINT "10 DIM M%(500):FOR I=770 TO 790:READ
      J:POKE I,J:NEXT"
5030  PRINT "20 READ N:FOR I=0 TO N:READ M%(I):NE
     XT"
5040  PRINT "50 FOR I=0 TO N:POKE 768,M%(I):POKE
     769,30: CALL 770"
5043  PRINT "60 FOR J=1 TO D:NEXT:NEXT"
5045  PRINT "30 HOME:VTAB 10:?"Q$"SPEED:"Q$";:INP
     UT "Q$Q$";W$:D=VAL(W$)"
5050  PRINT "100 DATA 173,48,192,136,208,5,206,1,
     3,240,9,202,208,245,174,0,3,76,2,3,96"
5055  PRINT "105 DATA " STR$ (L)
5060 D = 110:I = 0
5070 X$ =  STR$ (D) + " DATA "
5080 T$ =  STR$ (M%(I)): IF  LEN (T$) +  LEN (X$)
      >  = 240 THEN  PRINT  LEFT$ (X$, LEN (X$) -
     1):D = D + 10: GOTO 5070
5090 X$ = X$ + T$ + ",":I = I + 1: IF I = L + 1 THEN
     5110
5100  GOTO 5080
5110  PRINT  LEFT$ (X$, LEN (X$) - 1): PRINT "RUN
     ": PRINT D$"CLOSE": PRINT D$"EXEC "FL$
6000  HOME : END
```

10.7 MUSIC ENTRY

Program RECORDER lets you enter and store a song, and create a program with it, but does not allow you to copy music in written form. In program ENTRY the input is done with note names and durations, so that you can hear songs without knowing how to play.

Line 5 initializes some commonly used variables so that they do not have to be converted into floating point numbers, and the process is faster.

PIT is the location where the pitch has to be POKEd, DUR the duration, NI the value poked into DUR, AN is 1 (ONE cannot be used, since it is parsed as ON A), and ZE is 0.

Line 10 READs and POKEs the music routine, and line 20 DIMensions the arrays: N%( ) will keep the height of the notes, N$( ) the entries from the keyboard, and M%( ) the actual numbers to play. Line 30 READs the values corresponding to the notes, and line 35 shows the following options:

1-ENTER OR EDIT NOTES
2-PROCESS AND PLAY
3-SAVE TO DISK
4-LOAD FROM DISK
5-QUIT

1-ENTER OR EDIT NOTES

There are four components for each note:

1. The pitch (called here NOTE)
2. Whether it is natural (N), flat (b), or sharp (#)
3. Which octave it is in (from 1 to 3)
4. Its duration.

Line 50 PRINTs the appropriate titles. Line 60 PRINTs the number of the note we are at (the index of the array).

Subroutine 450 PRINTs the values currently in that position of the array: N$ (I, 0) is the note, N$ (I, 1) is the variation (#, b, N), N$ (I, 2) is the octave, and N$ (I, 3) is the duration.

If this note has not been entered, empty strings are PRINTed.

Line 70 GETs the first value. If W$ is the left arrow (CHR$ (8)) and C (the index of the array) is not at the very first location, the previous values are displayed; it is used to go backwards in the array.

Line 90 uses the right arrow (CHR$ (21)) to do the opposite: to go forward until the upper bound is found (N stores the highest value).

Lines 102 and 104 take the program back to the menu, not without adjusting N if C is higher than the current value of N, when more notes have been entered since its last update. If L is typed, line 120 POKEs 9 into 34 (to protect the titles from the scrolling that might occur), and lists all the notes stored so far (C is the index, and subroutine 450 displays the four items of each line). If the list does not fit in the screen, ctrl-S can be used to temporarily "freeze" the display.

Lines 130 to 150 pause until a <return> is pressed, POKE into 34 its usual value, and return to the note entry point in line 50.

Line 160 rejects any illegal notes, and line 170 sets N$ (C, 0) to W$ if the note is accepted; it prints it too, since GET doesn't echo the input. Lines 180 and 190 accept the note variations (#, b, N), and set N$ (C, 1) to it, echoing W$.

Lines 200 and 210 accept the octave and store it in N$ (C, 2), and lines 220 and 230 GET the duration of the note. C is incremented, and the loop is repeated.

2-PROCESS AND PLAY

When N = 0 (there are no notes to play), line 240 returns to the menu immediately. Otherwise, line 250 starts the loop that will process every note entered so far, from 0 to N. Lines 260 to 320 compare N$ (the name of the note) with the letters from C to A, and sets P to the corresponding value.

Notice that there are two semi-tones between C-D, D-E, F-G, G-A, and A-B, but not between E-F and B-C. Since the values in N% ( ) are stored in increments of a semi-tone, the values of P are not successive, but rather reflect this distance between notes.

Line 330 increases P by 1 if the variation is a # (by definition, # makes the note one semi-tone higher), or line 340 decreases P by 1 if it is a b; an N (normal) does not affect the pitch.

Since the distance between octaves is 12 (there are 12 semi-tones in each octave), the number of the octave is multiplied by 12 and added to P (the first octave starts at 1, therefore (1–1)*12 is added to it). Finally, lines 360 to 420 analyze the duration code and assign to V the corresponding value, to use it as a guide in a delay loop.

The duration codes are given in Table 10.2.

Table 10.2. Note duration codes.

CODE	EQUIVALENCE
1	whole
2	half
3	3/4 of a half (quarter with dot)
4	quarter
6	3/4 quarter (eighth with dot)
8	eighth
9	sixteenth

After all the notes have been converted into their numerical equivalent (doing it as they are played would delay the process incredibly), line 440 plays them, by POKEing the corresponding pitch and duration, and delaying (J FOR loop) for the appropriate amount of time.

No provision is made for variation of tempos, but working out different values for V (lines 360 to 420) is not too difficult. Multiplying these numbers by a scaling factor usually does not keep the correspondence (except when the factor is close to 1).

3-SAVE TO DISK

Line 500 goes to 800 for the filename (back to the menu if <return> is typed>), OPENs the file, and PRINTs N and the contents of N$ (I, J).

4-LOAD FROM DISK

Line 600 goes to 800 for the filename, and reads N$ (I, J).

```
1  REM  ** MUSIC ENTRY **
5 PIT = 768:DUR = 769:NI = 40:SV = 770:AN = 1:ZE =
     0
10  FOR I = 770 TO 790: READ J: POKE I,J: NEXT
20  READ N: DIM N%(N),N$(300,3),M%(300,1)
30  FOR I = 1 TO N: READ N%(I): NEXT :N = 0:D$ =
      CHR$ (4)
35  HOME : VTAB 8: PRINT "1-ENTER OR EDIT NOTES":
      VTAB 10: PRINT "2-PROCESS AND PLAY": VTAB 1
     2: PRINT "3-SAVE TO DISK": VTAB 14: PRINT "4
     -LOAD FROM DISK": VTAB 16: PRINT "5-QUIT ";
```

```
37  GET W$:V =  VAL (W$): IF V < 1 OR V > 5 THEN
     37
40  ON V GOTO 50,240,500,600,700
50  HOME : VTAB 4: PRINT "NOTE # ": INVERSE : VTAB
     8: HTAB 2: PRINT "NOTE";: HTAB 12: PRINT "#,
     F,N";: HTAB 22: PRINT "OCTAVE";: HTAB 32: PRINT
     "DURATION": NORMAL
60  VTAB 4: HTAB 8: PRINT C"  ": VTAB 10: GOSUB 4
     50
70  VTAB 10: HTAB 3: GET W$: IF W$ = "" THEN
80  IF W$ <  > CHR$ (8) OR C <  = 0 THEN 90
82  IF C > N THEN N = C
84 C = C - 1: GOTO 60
90  IF W$ =  CHR$ (21) AND C < N THEN C = C + 1: GOTO
     60
100  IF W$ <  >  CHR$ (27) THEN 110
102  IF C > N THEN N = C
104  GOTO 35
110  IF W$ <  > "L" THEN 160
115  IF C > N THEN N = C
120  VTAB 10: HTAB 1: CALL  - 958: POKE 34,9: FOR
     C = 0 TO N: GOSUB 450: NEXT
130  PRINT "TYPE <RETURN> TO CONTINUE ";
140  GET W$: IF W$ <  >  CHR$ (13) THEN 140
150 C = N: POKE 34,0: GOTO 50
160  IF W$ < "A" OR W$ > "G" THEN 70
170  PRINT W$;:N$(C,0) = W$
180  HTAB 14: GET W$: IF W$ <  > "#" AND W$ <  >
     "F" AND W$ <  > "N" THEN 180
190  PRINT W$;:N$(C,1) = W$
200  HTAB 24: GET W$: IF W$ < "1" OR W$ > "3" THEN
     200
210  PRINT W$;:N$(C,2) = W$
220  HTAB 35: GET W$:N$(C,3) = W$
230 C = C + 1: GOTO 60
240  IF N = 0 THEN 35
250  FOR I = 0 TO N:N$ = N$(I,0)
260  IF N$ = "C" THEN P = 1: GOTO 330
270  IF N$ = "D" THEN P = 3: GOTO 330
280  IF N$ = "E" THEN P = 5: GOTO 330
290  IF N$ = "F" THEN P = 6: GOTO 330
300  IF N$ = "G" THEN P = 8: GOTO 330
310  IF N$ = "A" THEN P = 10: GOTO 330
320  IF N$ = "B" THEN P = 12
330 B$ = N$(I,1): IF B$ = "#" THEN P = P + 1: GOTO
     350
340  IF B$ = "F" THEN P = P - 1
350 D =  VAL (N$(I,2)) - 1:P = P + (12 * D)
360 B$ = N$(I,3): IF B$ = "1" THEN V = 800: GOTO
     430
370  IF B$ = "2" THEN V = 600: GOTO 430
380  IF B$ = "3" THEN V = 370: GOTO 430
```

```
390   IF B$ = "4" THEN V = 200: GOTO 430
400   IF B$ = "6" THEN V = 125: GOTO 430
410   IF B$ = "8" THEN V = 70: GOTO 430
420   IF B$ = "9" THEN V = 0
430  M%(I,0) = P:M%(I,1) = V: NEXT
440   FOR I = 0 TO N - 1: POKE PIT,N%(M%(I,ZE)): POKE
      DUR,NI: CALL SV: FOR J = AN TO M%(I,AN): NEXT
      : NEXT : GOTO 35
450   HTAB 3: CALL  - 958: PRINT N$(C,0);: HTAB 14
      : PRINT N$(C,1);: HTAB 24: PRINT N$(C,2);: HTAB
      35: PRINT N$(C,3): RETURN
500   GOSUB 800: PRINT D$"WRITE "W$: PRINT N: FOR
      I = 0 TO N: FOR J = 0 TO 3: PRINT N$(I,J)
510   NEXT : NEXT : PRINT D$"CLOSE": GOTO 35
600   GOSUB 800: PRINT D$"READ "W$: INPUT N: FOR I
       = 0 TO N: FOR J = 0 TO 3
610   INPUT N$(I,J): NEXT : NEXT
620   PRINT D$"CLOSE": GOTO 35
700   HOME : END
800   VTAB 20: HTAB 3: INPUT "FILENAME:";W$
805   IF W$ = "" THEN POP : GOTO 35
810   PRINT : PRINT D$"OPEN "W$: RETURN
1000   DATA  173,48,192,136,208,5,206,1,3,240,9,20
      2,208,245,174,0,3,76,2,3,96
1500   DATA  48
2000   DATA  255,240,228,215,204,193,185,171,162,1
      53,145,136,128
2100   DATA 120,114,107,102,96,92,85,81,76,72,68,6
      4
2200   DATA 60,56,53,50,48,45,42,40,37,35,33,31
2300   DATA 29,28,25,24,23,21,20,19,18,17,16,15
```

10.8 HARMONY

So far, we have been limited to playing one note at a time, because the music routine is made that way. To play in harmony (more than one note simultaneously) we would need a different and very complicated music routine, since playing two notes (or more) require dealing with the frequencies of both notes, and how they affect each other.

Fortunately, Applesoft is fast enough to produce notes in succession, fast enough to make us believe we are hearing them simultaneously. Actually, there is a vibrato-like effect (since the notes are being produced in succession) which becomes more noticeable as more notes are added to the harmony.

Program BELLS produces three different sounds in harmony. Line 10 loads the usual music routine, and line 15 initializes the variables PIT and DUR to the locations where the pitch and durations must be POKEd, NI to the duration used. Notice that in previous programs a value of 40 or higher was being used. Here, since the notes must be produced in a quick succession, a much shorter note is needed (9 is poked into DUR); smaller numbers tend to make the sound less musical and more like a noise.

Line 40 READs the musical equivalences into N ( ), and line 42 assigns some values to variables A-K. These values are the actual indices of the pitches desired, already in variables, for speed.

Line 44 does the first sound, a telephone-like ring. The values POKEd into the pitch byte correspond to a C and an F# of the highest octave.

Line 46 produces a chord with seven notes. Such a big number makes the non-simultaneity of the notes very noticeable, producing a sound that resembles those in some arcade games.

Line 48 uses numbers as indices in N ( ) to produce still another different sound, and the cycle is repeated starting at line 43.

```
1   REM   ** BELLS **
10   FOR I = 770 TO 790: READ J: POKE I,J: NEXT
15 AN = 768:TW = 769:DU = 9:D1 = 200:SS = 770:D2 =
     10
30   READ N: DIM N(N)
40   FOR I = 1 TO N: READ N(I): NEXT
42 A = 48:B = 35:C = 255:D = 171:E = 136:F = 128:
     G = 114:H = 102:K = 85
43   FOR J = 1 TO 5
44   FOR I = 1 TO 15: POKE AN,A: POKE TW,DU: CALL
     SS: POKE AN,B: POKE TW,DU: CALL SS: NEXT
45   FOR I = 1 TO 400: NEXT : NEXT
46   FOR I = 1 TO 10: POKE AN,C: POKE TW,D2: CALL
     SS: POKE AN,D: POKE TW,D2: CALL SS: POKE AN,
     E: POKE TW,D2: CALL SS: POKE AN,F: POKE TW,D
     2: CALL SS
47   POKE AN,G: POKE TW,D2: CALL SS: POKE AN,H: POKE
     TW,D2: CALL SS: POKE AN,K: POKE TW,D2: CALL
     SS: NEXT
48   FOR I = 1 TO 20: POKE AN,N(20): POKE TW,DU: CALL
     SS: POKE AN,N(24): POKE TW,DU: CALL SS: POKE
     AN,N(27): POKE TW,DU: CALL SS: NEXT : GOTO 42
```

```
50   FOR I = 1 TO N: POKE 768,N(I)
60   POKE 769,90: CALL 770
65   NEXT
1000  DATA  173,48,192,136,208,5,206,1,3,240,9,20
     2,208,245,174,0,3,76,2,3,96
1500  DATA   48
2000  DATA  255,240,228,215,204,193,185,171,162,1
     53,145,136,128
2100  DATA 120,114,107,102,96,92,85,81,76,72,68,6
     4
2200  DATA 60,56,53,50,48,45,42,40,37,35,33,31
2300  DATA 29,28,25,24,23,21,20,19,18,17,16,15
```

10.9 SONGS IN HARMONY

Program CHORDS uses an array to store the three notes of every chord so that they can be played or SAVEd using DATA.

Line 5 initializes the usual variables, this time with TM = 10.

Line 10 READs and POKEs the music routine. Lines 30 and 40 DIMension and READ the musical equivalences. M% (I, J) will store the notes of each chord in M% (I, 1), TM% (I, 2), and M% (I, 3).

The notes are READ in lines 42 and 43.

Lines 48 and 49 play these notes with I counting the chord (first chord, second . . .), K is used to play each group of three notes five times, and J is used as an index for the actual notes of the chord.

To find the notes for a particular song, they must be harmonized in three or less notes (when three are not available in some parts, just duplicate the melody), their corresponding values found in the music equivalence table, and their values stored as DATA.

```
1  REM  ** CHORDS **
5 FS = 768:SC = 769:SS = 770:DU = 10
10  FOR I = 770 TO 790: READ J: POKE I,J: NEXT
30  READ N: DIM N(N),M%(100,3)
40  FOR I = 1 TO N: READ N(I): NEXT
42  FOR I = 1 TO 35
43  FOR J = 1 TO 3: READ M%(I,J): NEXT : NEXT
48  FOR I = 1 TO 35: FOR K = 1 TO 5
49  FOR J = 1 TO 3: POKE FS,N(M%(I,J)): POKE SC,D
     U: CALL SS: NEXT : NEXT : NEXT
1000  DATA  173,48,192,136,208,5,206,1,3,240,9,20
     2,208,245,174,0,3,76,2,3,96
1500  DATA    49
```

```
2000  DATA  255,240,228,215,204,193,185,171,162,1
    53,145,136,128
2100  DATA 120,114,107,102,96,92,85,81,76,72,68,6
    4
2200  DATA 60,56,53,50,48,45,42,40,37,35,33,31
2300  DATA 29,28,25,24,23,21,20,19,18,17,16,15
3000  DATA  5,8,13,6,10,15,8,13,17,13,17,20,15,18
    ,20,13,18,22,13,17,20,8,13,17,5,8,13,8,12,15
    ,8,13,17,8,1,17,8,15,15,5,8,13
3010  DATA 3,10,15,6,10,15,5,8,13,6,10,15,8,13,17
    ,13,17,20,13,17,20,13,18,22,13,17,20,8,13,17
    ,5,8,13,8,12,15,8,13,17,8,13,17
3020  DATA 8,13,15,8,12,15,6,8,13,3,8,13,1,8,13,1
    ,8,13,1,13,25,1,13,25
```

UTILITIES

11.1 BOOLEAN VARIABLES

Just as the string variables can store characters, the real variables can store real numbers and the integer variables can store integer numbers. A fourth type, boolean, can be stored in either a real or an integer variable. It has its own operations and behaves in a different way than the other three.

A boolean value can be the result of two operations: a comparison or a boolean operation. Typical examples of the first are

X <= 23 and M (I) <> S;

examples of the boolean operations would be

A AND B and (X > B) AND (12 <= E).

The only two possible answers to a comparison are TRUE or FALSE; that is why in the IF statements only two possibilities are considered. If the comparison is TRUE, the instructions following the IF statement are executed; if it is FALSE, execution continues in the next line number.

The basic concept of boolean variables is that they can only take either one of the two values: TRUE or FALSE, no matter how complicated the operations or expressions involved.

Here is an example of an equivalence that is not so obvious:

```
10 IF X > 32 THEN . . .
```

is equivalent to:

```
10 B = X > 32 : IF B THEN . . .
```

In the second example, the result of the comparison is stored in a variable, and then the content of B is used in the IF statement. The behavior is exactly the same in both cases. Whenever there is a comparison in Applesoft, the result is assigned either 1 or 0; 1 is considered as TRUE, and 0 as FALSE.

Apart from the relational symbols (e.g., =, <>, <=), there are three boolean operators in Applesoft: AND, OR, and NOT.

Table 11.1 shows their behavior.

Table 11.1. Boolean operators.

AND operator:

X	Y	X AND Y
FALSE	FALSE	FALSE
FALSE	TRUE	FALSE
TRUE	FALSE	FALSE
TRUE	TRUE	TRUE

OR operator:

X	Y	X OR Y
FALSE	FALSE	FALSE
FALSE	TRUE	TRUE
TRUE	FALSE	TRUE
TRUE	TRUE	TRUE

NOT operator:

X	NOT X
TRUE	FALSE
FALSE	TRUE

The priority of the boolean (or logical) and relational operators is as follows:

NOT
>, <, =, >=, =>, <=, =<, <>
AND
OR

Examples of Boolean Functions

FUNCTION ODD

Returns TRUE if the argument is an odd number; otherwise returns FALSE.

```
10 DEF FN ODD (X) = INT (X/2) = X/2
20 INPUT N : IF FN ODD (N) THEN ?"ODD " : GOTO 20
30 ?"EVEN"
```

FUNCTION EVEN

Uses the previous function (odd), plus the operator NOT, to return TRUE when the argument is even.

```
10 DEF FN ODD (X) = INT (X/2) = X/2
20 DEF FN EVEN (X) = NOT FN ODD (X)
30 INPUT N : IF FN EVEN (N) THEN ?"EVEN" : END
40 ?"ODD"
```

FUNCTION MAXIMUM

```
10 FUNCTION MAX (X) = (X > Y) * X + (Y > X) * Y
20 INPUT A, Y : ? FN MAX (A)
```

When A > Y, the parenthesis in (X > Y) evaluates as TRUE (1), which multiplied by X leaves X, and (Y > X) evaluates as FALSE (0), which eliminates Y; the result is X, the maximum of the two. When the order is reversed, Y is returned as the maximum.

```
10 FUNCTION MIN (X) = (X < Y) * X + (Y < X) * Y
20 INPUT W, Y : ? FN MIN (W)
```

FUNCTION LEGAL

This function can be used to see if a number is a valid parameter to use in HCOLOR. By changing the numbers compared, it can be used to test for inclusion in a certain interval.

```
10 DEF FN LEGAL (X) = X >= 0 AND X <= 7
20 INPUT C : IF NOT FN LEGAL (C) THEN ?"ILLEGAL HCOLOR " :
GOTO 20
```

FUNCTION DAY

Returns TRUE if the date (the name DATE cannot be used as the function's name because of the embedded AT in it) in X$ is valid.

X$ must be in the format MM/DD/YY, since the month, day, and year will be looked for in those locations. Notice that, since a string cannot be passed as the argument of a function, the dummy argument X is used.

```
10 DEF FN DAY (X) = (VAL (LEFT$ (X$, 1)) > 0 AND VAL (LEFT$
(X$, 1)) < 32) AND (VAL (MID$ (X$, 4, 2)) > 0 AND VAL (MID$
(X$, 4, 2)) < 13) AND VAL (RIGHT$ (X$, 2)) > 0
20 INPUT"TODAY'S DATE (MM/DD/YY) ";X$ : IF NOT FN DAY (1)
THEN PRINT"ILLEGAL DATE " : GOTO 20
```

11.2 RECOVERING FROM "NEW"

When Applesoft executes the command NEW, it erases the program in memory; if the command was issued by mistake and you didn't have a copy of the program, you might think that there is no way to bring it back.

If you follow the procedure described below, the program can be recuperated in its entirety.

What has really happened is that Applesoft has changed the values of some pointers, but the program is still intact in memory; if the pointers are set back to the values they had before the NEW, the program will be just as good as it was before.

The first thing to do is to POKE 2050, 8; this points to the first instruction, which will reappear intact if you type LIST.

Applesoft needs the distance to the next line number, and in this case you will have to calculate how far the next instruction is. This can be done by counting the characters in the first instruction, remembering that every Applesoft reserved word (FOR, PRINT, etc.) is stored as one character. Add 6 to the number of characters you think there are in the first instruction, and POKE that value into location 2049.

LIST the program. If it doesn't look correct, try POKEing into 2049 values around the one you calculated. Sometimes it is hard to come up with the exact number, because of extra spaces added by Applesoft.

When the program is finally LISTing correctly, there are still two pointers to adjust: locations 105 and 106 (least significant byte and most significant one, in that order) which point to the start of the simple variable space, and locations 175 and 176 (least and most significant, as before), which point to the end of the program.

Since we don't know exactly where in memory the program ends (which usually coincides with the point where the variable space begins, unless changed by LOMEM:), we might POKE two values large enough to be sure that the program will not be cut in half (if you set the end of the program and the beginning of the space for the variables in an address which is lower than the actual end of the program, when the value of any variable is changed, part of the program is affected too).

If your computer has 48k of memory, it is perfectly safe to assume that the program ends at location 30000, because it would have to be a huge program to extend past this address. But we want to be sure, because once it is destroyed, there is no way to recuperate it.

Using a calculator (program CONVERSION in section 11.13 can calculate the bytes, but in this case nothing can be put in memory before the program is completely recuperated. Study it, anyway, to understand how the conversion is made), we find that 30000 corresponds to 48 and 117 (least and more significant bytes).

If you POKE 48 into 105 and 175, and 117 into 196 and 176, the program runs perfectly.

If you try to run it or even use a variable before resetting these pointers (like trying to do the calculations with the computer, using some variables), Applesoft thinks the memory is all free, places the variables at the beginning of the memory, and destroys part of the program by writing over it.

Even though the program runs correctly, it has very little free memory, and if it has a big DIM or uses many variables, it will run out of memory. Besides, when it is SAVEd to the disk, the pointers indicate that the program expands until location 30000, and all of it is SAVEd, occupying a big part of the disk.

To solve this, we can make use of the EXEC command.

Add the following lines anywhere in the program, choosing line numbers that are not used (line 0 is usually a good choice).

```
0 POKE 33, 33 : D$ = CHR$ (4) : ?D$"OPEN DUMMY" :
?D$"WRITE DUMMY" : LIST : ?D$"CLOSE" : END
```

Running this miniprogram creates a text image of the program without pointers of any kind. The reserved words of Applesoft are no longer in their condensed form, but in pure ASCII characters. This happens because when a file is OPENed to WRITE, anything that would normally go to the terminal ends up in the disk (even ERROR messages and the line numbers of a TRACE). The POKE 33, 33 makes all the extra spaces added after line 33 disappear, just like when a line is being edited.

Now that the ASCII copy of the program is in the disk, type NEW and EXEC DUMMY (or the name used to store it). Then, you will have a perfect copy of your original program, plus the line or lines added to SAVE it, which should be deleted. Then, the program can be SAVEd and treated as any other Applesoft program.

This technique is also very helpful when part of a program becomes unreadable, or some lines undeletable.

Sometimes ghost line numbers appear (fortunately not too often, if your computer is working correctly), and it is impossible to delete them or RUN the program.

SAVEing it to the disk saves the bad lines too, but the image will not have any unnatural pointers. So, add the same instructions used above, do an ASCII copy of the program in disk, EXEC it, and delete the unwanted lines. Some of the program might be lost, but not all (hopefully).

11.3 CREATING HIDDEN COMMANDS

Three very important characters in the Apple cannot be normally included or used in lines of a program:

1. <return>, CHR$ (13) or ctrl-M.

   Since this character is used as the terminator of all inputs including instructions and program lines, as soon as you press it, Applesoft starts working with the line just entered to store, execute, or discard it, but the CHR$ (13) never forms part of the characters typed.
2. Backspace, CHR$ (8), or ctrl-H.

   This character is used to delete one character. Therefore, whenever Applesoft receives it, instead of adding it to the characters already entered, it deletes the last one received.

3. Ctrl-D or CHR$ (4) as a DOS command.

   DOS only recognizes this character when it is preceded by a <return>. Whenever a PRINT is executed without a semicolon at the end of the variables or constants, it automatically adds a return, and since <return>'s cannot be normally included in a line, ctrl-D's do not turn ON disk commands.

When the right arrow (CHR$ (21)) goes over a character on the screen, Applesoft receives it and treats it just as if it had actually been typed.

The control character alphabet expands from ASCII 1 to 26, and the INVERSE mode POKEs the characters in the video memory with exactly those codes. In other words, when you go over an inverse A, for example, Applesoft interprets it as if you had entered a ctrl—A, without taking the action that such a character would cause when typed directly.

That way, if you go over five inverse H's, what actually happens is that five backspaces enter, and are not interpreted as an attempt to delete anything. Since the backspace has the property of making the cursor move to the previous position on the screen, being able to insert it into program lines or strings gives you a great deal of control over the display.

Let's see an example. Type the following:

```
INVERSE : ?"HHHHHHHHHH" : NORMAL <return>
```

The ten H's in INVERSE appear on the screen. Now, type

```
10 REM
```

and before pressing <return> to finish the line, move the cursor up (using the <esc> codes) to the beginning of the H's. Go over them with the right arrow and when you come to the end, type ten spaces, and the words INVISIBLE LINE. Press <return>.

If you did things right, when you say LIST, the title INVISIBLE LINE will appear, without a line number and without the REM; since it is a REM instruction, it does not interfere with the program in any way, and can be left there.

Since the line number cannot be seen, a good deal of guess work has to be done to erase it. The only way to see the line number and the REM is to set SPEED to a low value. Do this to see in slow motion how the trick is done.

The same techniques will be used now to add <return>'s to program lines. Type the following:

```
INVERSE : ?''MMMMMMMMMM'' : NORMAL
```

Then type

```
10 REM PROGRAM
```

then, before ending with <return>, move the cursor to the INVERSE M's; go over them with the right arrow, and type some spaces so that the following word is centered in the screen; type

```
MENU
```

go back and read some of the M's, leave with <esc>, and finally type

```
SEPARATOR
```

If you type LIST, you will see that every INVERSE M you read causes a jump to the next line; an attempt to edit such a line never works, since there are far more spaces between words than a line can take.

To go from PROGRAM to MENU, in the previous example, another technique would have worked just as well: Instead of using inverse M's, a ctrl-J (line feed) could be typed from the keyboard. This makes the cursor jump to the next line, with the difference that the cursor does not go to the left margin of the screen. Add a number of backspaces (INVERSE H's) to put the cursor in the right position, and some spaces might be economized.

11.4 A VERY POWERFUL HIDDEN TOOL

All the disk commands in Applesoft must be preceded by a ctrl-D, which in turn must be preceded by a carriage return (CHR$ (13)), or else it is ignored, and the command is PRINTed on the screen, without any effect. When a <return> is followed by a ctrl-D in a line number and that line is LISTed, Applesoft interprets it exactly as if a PRINT had been typed because, at the end, LISTing is printing an instruction to the screen, and when the <return> ctrl-D sequence is encountered, Applesoft believes that what comes next is a disk command, and tries to execute it.

Let's see an example. Type the following:

```
INVERSE : PRINT "MD" : NORMAL
```

Now type

```
20 HOME : FOR I = 1 to 300 : ?"SO FAR, SO GOOD ! "; : NEXT
10 REM THIS IS AN UNLISTABLE PROGRAM
```

and before pressing <return>, use the <esc> codes to go to the inverse MD and read it using the right arrow. Right after the D, print FP and type <return>.

Before LISTing it, RUN it. It must run normally, because a REM instruction is always ignored. Now try LISTing it. Line 10 was listed, but not line 20; besides, if you try to list it now, you discover that there is nothing in memory!

What has happened is that as soon as a ctrl-D was printed (and it was preceded by a <return>), Applesoft understood that a DOS command was coming. It found FP, and thought it was the DOS command to go to floating point Basic (Applesoft), which, among other things, erases any program in memory. It can still be recuperated, and now you know how.

If you plant some of those REMs in your program, any attempt to LIST it will only go as far as the first REM encountered. But more can be done: Where you said FP, you could have typed RUN LEMONADE, for example. This way, instead of listing the program, a completely different action takes place. You could say CATALOG or any DOS command, even INIT ! (beware).

11.5 HIDING CHARACTERS IN THE CATALOG

The same technique used to hide characters in REM statements can be used to put them in any other instruction; the (optional) LET is a very useful one.

Program HIDDEN CATALOG combines EXEC files with hidden characters to give you a protection of your disk files which is very hard to break.

Line 10001 assigns a quote (CHR$ (34)) to Q$; lines 10003 through 10005 do the INPUT of the line number, character, and program name. Line 10007 OPENs the file DUMMY and DELETEs it.

DUMMY will be used to store a short EXEC file. When EXECed, every character in it will be read and acted upon. Since WRITEing to an already existing file doesn't erase its previous contents, any characters after those written this time will still be EXECuted, causing unexpected results. If, on the other hand, DELETE is issued and the file does not exist, a FILE NOT FOUND error causes termination of the program; that is why it is OPENed before it is DELETEd. Now we are left with a smaller possibility of error, which is when a Dummy file of different type (program or binary) would cause a FILE TYPE MISMATCH error when attempting to OPEN it. This can only be prevented with an ONERR GOTO routine.

Lines 10008 and 10009 reOPEN DUMMY, this time to use it. Lines 10010 through 10014 PRINT the following program fragment into DUMMY. Let's assume 123 is the line number, H the initial character, and TEST the program name:

```
123 A$="H               "
DEL 10000, 10016
RUN 10017
```

There are exactly 15 spaces between the H and the quote.

Line 10015 CLOSEs the file and 10016 EXECs it. It is convenient to put an END right after the EXEC instructions since sometimes its absence bombs the EXEC command.

After the first two lines are EXECed, the program has a new form:

```
123 A$="H               "
10017 FOR I = 2058 TO 2065 : POKE I, 8 : NEXT
10018 DEL 10017, 10018
```

The last command in the EXEC file, RUN 10017, makes the I-loop POKE eight backspaces exactly in the area of memory where the spaces reside, right after the H in instruction 123, which is the only one left in memory after line 10018 is executed.

To find the exact address where this ctrl-H's had to be POKEd, the memory was scanned starting at location 2048, where the program usually starts (unless relocated). See Section 11.7, using the simple instruction:

```
FOR I = 2048 TO 3000 : PRINT I, CHR$ (PEEK (I)) : NEXT .
```

Now inside the quotes are one H, eight ctrl-H's, and seven spaces. If this string (available in A$ when the program is run) is used as a filename, the effect is the following: the H at the beginning of the string makes it a legal filename, since the first character of a filename must be a letter. The eight ctrl-H's make the cursor move left until all the letters that usually appear in the directory are to the right of it (the asterisk that denotes a LOCKed file, the type of file, and the number of sectors). The spaces delete those characters, since no two characters can be in the same position, and the actual filename provided is PRINTed.

The result is that this file looks exactly as any other one in the catalog, except for the information to the left, which it lacks completely.

Since ctrl-H is impossible to see (even with the control character finder routine in page 151 of the DOS manual), the file cannot be DELETEd, RENAMEd, or changed in any way, unless the right filename is provided, and it is not the one displayed on the catalog.

Furthermore, since a letter is already at the beginning of the filename, what actually shows on the screen doesn't need to be a string of letters, and doesn't even have to start with a letter. Using program HIDDEN CATALOG to SAVE your programs and files, it makes perfect sense to think of a catalog like this:

```
DISK VOLUME 254
*************
*           *
*  PROGRAMS *
*     BY    *
*   GABRIEL *
*   CUELLAR *
*           *
*************
```

Since the CATALOG command PRINTs the filenames directly in the screen, it is not affected by SPEED, and there is no way to see how the trick was done.

```
10000  REM  ** HIDDEN CATALOG **
10001 D$ =  CHR$ (4):Q$ =  CHR$ (34)
10002  HOME
10003  PRINT : INPUT "LINE NUMBER:";LN$
10004  PRINT : INPUT "INITIAL CHARACTER:";I$
10005  PRINT : INPUT "PROGRAM NAME:";N$
10006  IF W$ =  CHR$ (13) THEN 10007
10007  PRINT : PRINT D$"OPEN DUMMY": PRINT D$"DEL
     ETE DUMMY"
10008  PRINT D$"OPEN DUMMY"
10009  PRINT D$"WRITE DUMMY"
10010  PRINT LN$;" A$=";Q$;I$;: FOR I = 1 TO 15: PRINT
     " ";: NEXT
10011  PRINT N$;Q$
10013  PRINT "DEL 10000,10016"
10014  PRINT "RUN 10017"
10015  PRINT D$"CLOSE"
10016  PRINT D$"EXEC DUMMY": END
10017  FOR I = 2058 TO 2065: POKE I,8: NEXT
10018  DEL 10000,10018
```

Inserting Inverse of Flashing Characters in the Catalog

If the filename is formed reading INVERSE or FLASHing characters in direct mode as was done in Section 11.4, the result is not the same as when it is done from within a program. The filename is left in INVERSE or FLASHing characters and appears that way in the CATALOG.

Suppose you have a file called Hologram. To make it show in INVERSE, do the following. Type

INVERSE : ?“HOLOGRAM” : NORMAL

Then type

RENAME HOLOGRAM,

and use the <esc> codes to read the INVERSE “HOLOGRAM” PRINTed by the first instruction. Read it with the right arrow, and press <return>. Now CATALOG displays it in INVERSE mode.

11.6 HIDDEN CHARACTER CREATOR

Program HIDDEN CHARACTER CREATOR does what we have done already in direct mode: It creates a program line with a number that does not show, letting you type and edit the backspaces.

Line 10001 initializes D$ to be used in the disk operations and Q$ to the quote. Lines 10003 and 10004 ask for a line number and a variable name. Since a string variable is needed, a "$" sign is added if it is not provided.

Line 10005 PRINTs a numbered line that will make easier the positioning of the part of the line that is to be PRINTed (as opposed to the part that will be hidden). Line 1006 PRINTs the line number, the variable name, and the first quote in exactly the same way they will appear in the program. Line 10007 GETs a character; if it is <esc> (CHR$ (27)), it empties X$ (which is used as a buffer to store all the characters previously typed) and returns to line 10005. This way, you can reposition the title you want PRINTed (center it, for instance), because the backspace cannot be used here; if typed, it will become part of the line.

If <return> is typed, line 10008 jumps to 10011, indicating the end of the input. Otherwise, line 10009 PRINTs the character received, adds it to X$, and returns to 10007.

Line 10011 OPENs and DELETEs the file DUMMY. Lines 10012 and 10013 reOPEN it and set it to WRITE. Line 10014 PRINTs the line number specified, the variable name, the equal sign, and the quote, but instead of sending X$, it sends as many asterisks as characters are in X$, and finally a quote.

Line 10015 PRINTs the instruction

```
10031 FOR I = 2057 to XXXX,
```

where XXXX is the position in memory where the last quote is going to be. Lines 10016 and 10017 add the two lines:

```
DEL 10000, 10025
RUN 10026
```

Line 10018 CLOSEs DUMMY. Line 10020 opens a new file, DUMMY1, and line 10021 PRINTs into it the ASCII code of every character in X$, preceded by the number of characters. Line

10022 CLOSEs DUMMY1 and line 10023 EXECs it (notice the END after the EXEC).

Suppose that 123 was the line number typed, A$ was the variable, and X$ had 30 characters. Right after the first instruction of DUMMY DELetes lines 10000, 10025, the program in memory looks like this:

```
123 A$ = "********************
          **********"
10026 D$ = CHR$ (4) : K = 1
10027 PRINT D$"OPEN DUMMY": PRINT D$"READ DUMMY1"
10028 INPUT N:DIM N (N)
10029 FOR I = 1 TO N: INPUT N (I) : NEXT
10030 PRINT D$"CLOSE"
10031 FOR I = 2057 TO 2087
10032 POKE I, N (K) : K = K + 1
10042 NEXT
10043 DEL 10026, 10043
```

The next instruction in the EXEC file RUNs the program starting at line 10026, which reads the ASCII codes stored in DUMMY1, and POKEs them into memory starting at location 2057 which is where the first asterisk is. The last character is POKEd in top of the final quote of line 123. This does not really matter, since a quote is needed to terminate a string constant only if it is not at the end of a program line. That way, the string left in line 123 in this example corresponds exactly to the original contents of X$ and when the line is LISTed, no final quote is seen.

Line 10043 leaves in memory only the finished line.

```
10000  REM  ** HIDDEN CHARACTER CREATOR **
10001 D$ =  CHR$ (4):Q$ =  CHR$ (34)
10002  HOME
10003  PRINT : INPUT "LINE NUMBER:";LN$
10004  PRINT : INPUT "VARIABLE:";VA$: IF  RIGHT$
     (VA$,1) <  > "$" THEN VA$ = VA$ + "$"
10005  PRINT : PRINT "1---5---10----5---20----5--
     -30----5---40";
10006  PRINT : PRINT LN$;" ";VA$;" = ";Q$;
10007  GET W$: IF W$ =  CHR$ (27) THEN  PRINT :X$
      = "": GOTO 10005
10008  IF W$ =  CHR$ (13) THEN 10011
10009  PRINT W$;:X$ = X$ + W$: GOTO 10007
```

```
10010 X$ = X$ + W$: GOTO 10007
10011  PRINT : PRINT D$"OPEN DUMMY1": PRINT D$"DE
     LETE DUMMY1"
10012  PRINT D$"OPEN DUMMY1"
10013  PRINT D$"WRITE DUMMY1"
10014  PRINT LN$;" ";VA$;" = ";Q$;: FOR I = 1 TO
      LEN (X$) - 1: PRINT "*";: NEXT : PRINT Q$
10015  PRINT "10031 FOR I=2057 TO "; STR$ (2056 +
      LEN (X$));":C=PEEK(I)"
10016  PRINT "DEL 10000,10025"
10017  PRINT "RUN 10026"
10018  PRINT D$"CLOSE"
10019  PRINT D$"OPEN DUMMY2": PRINT D$"DELETE DUM
     MY2"
10020  PRINT D$"OPEN DUMMY2": PRINT D$"WRITE DUMM
     Y2"
10021  PRINT  LEN (X$): FOR I = 1 TO  LEN (X$): PRINT
      ASC ( MID$ (X$,I,1)): NEXT
10022  PRINT D$"CLOSE"
10023  PRINT D$"EXEC DUMMY1": END
10024  PRINT C, CHR$ (C)
10025  NEXT
10026 D$ =  CHR$ (4):K = 1
10027  PRINT D$"OPEN DUMMY2": PRINT D$"READ DUMMY
     2"
10028  INPUT N: DIM N(N)
10029  FOR I = 1 TO N: INPUT N(I): NEXT
10030  PRINT D$"CLOSE"
10032  POKE I,N(K):K = K + 1
10042  NEXT
10043  DEL 10026,10043
```

11.7 RELOCATING A PROGRAM

Every time a program is brought into memory, either by LOADing it, or RUNning it from disk, or typing it directly, it is placed right after the end of page one of the text memory, starting in location 2048. When LOMEM is changed, it changes the lower bound of the space for variables and not the actual program.

A pair of locations indicate in Applesoft where is the beginning of the program; by default they are pointing at position 2049. These locations are 103 and 104, in the usual sequence: least significant byte in 103 and most significant in 104.

If a new address is POKEd into these positions, Applesoft will believe that the program starts at the point indicated by this new location in memory.

Program RELOCATE asks in line 10 for a program name; that program will be LOADed and run after the program start pointers are reset, so that the new program is placed in the desired address.

Line 20 asks for a starting address, and line 30 calculates the most significant digit as explained in section 11.13, part 6. Line 40 finds the least significant byte, and lines 60 and 70 POKE these values into locations 103 and 104.

Apart from instructing Applesoft about the address, a 0 must be POKEd in the byte previous to the starting address.

Now the program can be LOADed into this new location.

Any legal memory address can be used to place the program, even after the two high resolution pages. A particularly interesting address is the beginning of the video memory. If your program is not going to use the text or low resolution graphic area, it can be placed there; when it has finished its process, a HOME command will erase the video memory, leaving no program at all.

```
1  REM  ** RELOCATE **
5  HOME
10  VTAB 8: INPUT "PROGRAM NAME:";PN$: PRINT
20  INPUT "STARTING ADDRESS:";SA
30 MSB =  INT (SA / 256)
40 LSB = SA - 256 * MSB
60  POKE 103,LSB
70  POKE 104,MSB
80  POKE SA - 1,0
90  PRINT  CHR$ (4)"RUN "PN$
```

11.8 HAVING MORE THAN ONE PROGRAM IN MEMORY

Every time Applesoft finds a command that changes the natural sequence of the program (like a GOTO, GOSUB, ON var GOTO, ON var GOSUB, NEXT, RETURN, and IF . . THEN), it looks for the line number specified in the command, starting at the beginning of the program.

The pointers to the beginning of the program are locations 103 and 104, as described before, and remain unchanged even when LOMEM is reset. But if the contents of those pointers are changed by POKE instructions, as soon as Applesoft has to look for a particular line number to branch to, it starts its search at the address indicated by the values in these two locations.

Therefore, if a program is LOADed into the usual area (starting at 2048), the pointers set to a higher address and a new program loaded in this new address, the old one is not changed at all.

If the second program is running and it changes the pointers so that they point at the first one, any GOTO or similar instruction will transfer control to the corresponding line number in the first program LOADed.

There is one problem, however. Since the first program occupies a lower position in memory, all its variables will change the memory above it and there is a great chance that the second program will be changed or deleted. However, if at the very beginning Applesoft is instructed that the memory available for variables starts at an address higher than the end of the second program, both programs can cohabit perfectly.

If LOMEM is higher than both programs and one of them calls the other by changing the program start pointers, the variables will still be intact, and will be shared by both programs. On the other hand, if when one program calls the other, it not only changes the starting address of the program (locations 103 and 104) and the pointers to the end of the program (175 and 176), but the limits of the memory too (LOMEM is in 105 and 106 and HIMEM in 115 and 116), each program will have its own memory, and will behave nearly as if it had a computer of its own.

The only additional piece of information needed is how much memory is left in each part: This resides in locations 112, 113, and 114, which must be stored before calling each other and must be reset on entrance.

This technique opens great possibilities, but has many dangers: Losing track of only one of the pointers means complete disaster. For those curious enough, not only two but as many programs as wanted can be kept in memory simultaneously.

11.9 SAVING AN APPLESOFT PROGRAM AS A BINARY FILE

Since most of the information needed to run a program resides in locations 0 through the end of the program, BSAVEing this entire range will store not only an exact, working copy of the program, but its status; in other words, its condition when it is saved.

That way, if a program is running, is interrupted (either by ctrl-C or by a STOP) and BSAVEd, whenever it is BLOADed once more, a mere CONT will suffice to make it continue execution at the line number where it was interrupted.

The variables may be gone (especially the strings, which are stored starting at HIMEM), but if none had been used or if they are not needed anymore (or if not only the program is included in the memory range BSAVEd, but the beginning of the variable space too), the program will behave normally.

11.10 CONTROLLING THE RESET KEY

Whenever a program is running in Applesoft and the RESET key is pressed, execution stops and you are returned to command level. There are three locations which affect what RESET does when pressed.

The easiest way to avoid intrusion into your program with a RESET, is to POKE 1012, 0. That way, as soon as the RESET is pressed, Applesoft believes it has just been turned ON, and loads DOS (equivalent to typing PR#6 <return>), erasing any program in memory.

A more sophisticated way to control it is used in program RESET CONTROL: When the RESET key is pressed, the program can jump to a small routine especially available to handle that case.

After line 10000 has POKEd the small machine language program into memory and has performed the CALL, every time RESET is pressed, the program jumps to line 10, which PRINTs the message RESET, and waits for a key-press to jump back to line 10010.

You can program lines 10 through 10000 the way you want; after the RESET key is pressed it will be equivalent to having executed a RUN 10.

The second number in the DATA, 12, is the distance between the beginning of the program and the location to which RESET will make Applesoft jump. In this case it is 12, because there are 12 characters in the first line: one for the GOTO, five for the number 10000, and six which are always used to store the line number and other information.

If you want to add other instructions before the line 10, you must count every character in every new line, taking into account that the reserved words are stored in only one byte, add 6 to each line, and change the second number in the DATA list to this new distance.

```
1   GOTO 10000
10   POKE 103,1: CALL 1002: HOME : PRINT "RESET": GET
      W$: GOTO 10010
10000  POKE 1010,0: POKE 1011,3: CALL  - 1169: FOR
      I = 768 TO 774: READ W: POKE I,W: NEXT : DATA
       169,12,133,103,76,102,213
10010  HOME : FOR I = 1 TO 500: HTAB 19: VTAB 10:
       PRINT I: NEXT
10020  FOR I = 1 TO 400: IF  INT (I / 100) = I /
      100 THEN  PRINT  CHR$ (7);
10030  NEXT : GOTO 10010
20000  REM  ** RESET CONTROL **
```

11.11 VOLUNTARY GARBAGE COLLECTION

When Applesoft runs out of memory because of string garbage, it performs a "garbage collection," in which it rearranges the pointers to the strings and actually moves them in memory to get back the maximum amount of free space.

Depending on the actual amount of memory available, this process can take from just a few seconds to over two minutes.

If you are in the middle of a program and the computer suddenly "dies," it is likely that it is cleaning the memory. Every time a FRE ( ) instruction is encountered, Applesoft performs a garbage collection automatically before PRINTing the amount of memory left in bytes.

The following instruction tells you how much memory is still left, in groups of 256 bytes, without performing a garbage collection:

```
? PEEK (112)–PEEK (110)
```

When you want to know the exact number of bytes left, multiply the result of the following instruction, and add PEEK (111) to it, like this:

```
X = 256 * (PEEK (112)–PEEK (110)) + PEEK (111)
```

11.12 BINARY NUMBERS

Just as we use ten digits (0-9) to represent different values in the decimal system, the binary system uses only two. Most computers store and handle information using this system, because at the present state of technology it is easier to work with only two different states (ON or OFF, LOW or HIGH, 0 or 1) than with ten.

Even though Applesoft displays every digit in base 10, certain areas are understood better if the binary system is at least known.

In the decimal system a number is represented by one or more digits whose position indicates its magnitude in powers of ten.

As an example, decimal number 191 actually means:

```
  1 * 100, or 1 * 10 ^ 2
+ 9 *  10, or 9 * 10 ^ 1
+ 1 *   1, or 1 * 10 ^ 0.
```

In the binary system, instead of using powers of ten, powers of two are used; the decimal number 191 would be represented as

```
  0 * 256     (256 = 2 ^ 8)
+ 1 * 128     (128 = 2 ^ 7)
+ 0 *  64     ( 64 = 2 ^ 6)
+ 1 *  32     ( 32 = 2 ^ 5)
+ 1 *  16     ( 16 = 2 ^ 4)
+ 1 *   8     (  8 = 2 ^ 3)
+ 1 *   4     (  4 = 2 ^ 2)
+ 1 *   2     (  2 = 2 ^ 1)
+ 1 *   1     (  1 = 2 ^ 0)
 ---------
      191
```

Binary equivalent of 191 = 010111111

In the Apple computer (and in most of the systems of comparable size or power), the memory is divided into bytes, each one of which is a collection of eight bits: every bit corresponds to a binary digit, that is, it can take two values: 1 or 0.

If all the first eight binary digits are 1, the total adds up to 255; if every digit is 0, the sum is equal to 0.

This means that a byte can only take values that are between 0 and 255, that is, 256 different values.

To find the decimal equivalent of a given binary number, every digit is multiplied by 2 raised to the power of its position, taking position 0 as the rightmost and 7 as the leftmost one, as shown in Figure 11.1.

Binary number = 10011011

Decimal equivalent =

```
   1 * 2 ^ 7 = 128
 + 0 * 2 ^ 6 =   0
 + 0 * 2 ^ 5 =   0
 + 1 * 2 ^ 4 =  16
 + 1 * 2 ^ 3 =   8
 + 0 * 2 ^ 2 =   0
 + 1 * 2 ^ 1 =   2
 + 1 * 2 ^ 0 =   1
               ----
               155
```

Figure 11.1. Finding the decimal equivalent of binary 10011011.

To find the binary equivalent of a decimal number, perform successive integer divisions (discarding the fractional part) by powers of two. Figure 11.2 shows how the binary equivalent of decimal 61 is found.

```
61 DIV 128 = 0    61 -  0 = 61
61 DIV  64 = 0    61 -  0 = 61
61 DIV  32 = 1    61 - 32 = 29
29 DIV  16 = 1    29 - 16 = 13
13 DIV   8 = 1    13 -  8 =  5
 5 DIV   4 = 1     5 -  4 =  1
 1 DIV   2 = 0     1 -  0 =  1
 1 DIV   1 = 1     1 -  1 =  0
```

Binary representation of 61 = 00111101

Figure 11.2. Conversion of decimal 61 to its binary representation (DIV means integer division).

A succession of 0's and 1's might make a lot of sense to a computer, but in the human mind it has a sort of hypnotic effect:

When you see these two digits filling up an entire page, you cannot group them together and you lose track of their position and therefore their value. That is where the hexadecimal system plays an important role.

If we choose 16 as the base of our numeric system, instead of 10 or 2, numbers from 0 to 15 can be represented with only one digit; since we only have ten digits (from 0 to 9), the letters A-F are used to represent the missing values, as shown in Table 11.2.

Table 11.2. Decimal-hexadecimal equivalences.

Decimal - Hexadecimal

0 - 0
1 - 1
2 - 2
3 - 3
4 - 4
5 - 5
6 - 6
7 - 7

Decimal - Hexadecimal

8 - 8
9 - 9
10 - A
11 - B
12 - C
13 - D
14 - E
15 - F

The hexadecimal and the binary systems have something in common: Since both bases are powers of two, one hexadecimal digit can represent a binary number of four digits (usually called a "nibble"). Therefore, it only takes two hexadecimal digits to represent one byte, as shown in Table 11.3.

Table 11.3. Hexadecimal to binary equivalence.

HEXADECIMAL	BINARY	DECIMAL
00	00000000	0
01	00000001	1
02	00000010	2
03	00000011	3
.	.	.
.	.	.
FF	11111111	256

Hexadecimal digits are very often used in computer programming, especially when doing machine language programs, since they represent a byte very clearly.

Two Byte Representation

The maximum memory address in the Apple is the number 2 ^ 16, or 65535. Since most addresses are bigger than 255 (the maximum value of a byte), they are stored in two bytes, called the least and most significant ones. To find the two byte representation of a number, divide it by 256 (integer division), and you get the most significant byte; subtract from the original number the most significant byte multiplied by 256, and you get the least significant byte.

To find the address starting with the two bytes, multiply the most significant one by 256 and add it to the least significant byte.

11.13 CALCULATING EQUIVALENCES

Program CONVERSION does all of these calculations for you.

Except for the required initializations, all the line numbers are above 32000, so that you can load it and use it while working in another program (which hopefully does not have line numbers above 31999).

Line 10 sets TRUE and FALSE to their Applesoft equivalents. HX$ ( ) is DIMensioned to store in it the 16 binary digits, with its value corresponding to its index ("A" will be stored in position 10, "B" in 11, etc.).

A special note about the DATA line: You might have noticed that only the last letter (F) is enclosed in parentheses; this is because when strings are READ from DATA, the last one of the line gets a space after the character, and for purposes of comparison it would invalidate the "F" ("F" is not equal to "F "). If a DATA item is enclosed in quotes, it is READ exactly as it is.

Line 32000 displays the following menu:

```
1-DECIMAL TO BINARY
2-BINARY TO DECIMAL
3-DECIMAL TO HEXADECIMAL
4-HEXADECIMAL TO DECIMAL
5-DECIMAL ADDRESS TO BYTES
6-BYTES TO DECIMAL ADDRESS
7-QUIT
```

1-DECIMAL TO BINARY

Line 32040 calls subroutine 32380, which asks for a decimal number and discards it if it is outside the byte range (0-255). Subroutine 32400 waits until <return> is pressed to repeat the prompt for the number.

If the number is valid, line 32050 empties A$. It will be used to store the binary digits in string form, and at the beginning we do not want any interference from its previous contents.

The I-loop integer divides V by decreasing powers of two, leaving the result in S; the value of S times the current power of two is subtracted from V and S is added, in string form, to A$. Line 32060 PRINTs A$ and calls subroutine 32400 to wait for a <return> before returning to the menu.

2-BINARY TO DECIMAL

Line 32070 asks for a binary number and rejects it if it doesn't have exactly eight digits. If it does, line 32080 sets V (where the accumulated decimal value will be stored) to 0. The I-loop scans every digit (which is actually in string form), starting from the right (least significant bit). If the digit is not a 0 or a 1, an appropriate message is PRINTed and the number discarded; otherwise, it is multiplied by the current power of two in line 32090, and added to V. V is PRINTed and subroutine 32400 call to wait for a <return>.

3-DECIMAL TO HEXADECIMAL

Line 32100 calls subroutine 32380 for a decimal number. Line 32110 integer divides the number by 16, storing in S the most significant digit. To get the hexadecimal digit, HX$ ( ) is used. The least significant digit is found by subtrating S times 16 from V, and using the result as the index of HX$ ( ), which is added to A$. The result is PRINTed and subroutine 32400 is called for a pause.

4-HEXADECIMAL TO DECIMAL

Line 32120 asks for a hexadecimal number and rejects it if it does not have exactly two characters (two hexadecimal digits represent exactly one byte). Otherwise, line 32130 sets V (which will store the decimal value) to 0. The two characters in W$ are compared with every character in HX$ ( ). If they are found to be different from the 16 hexadecimal digits, a message is PRINTed and the number is rejected. Otherwise, the index of HX$ ( ) has the value of the digit, and line 32150 multiplies it by the proper power of 16, to add it to V. The result is displayed and subroutine 32400 is called for a pause.

5-DECIMAL ADDRESS TO BYTES

Line 32200 asks for a decimal number and rejects it if it is outside the legal range. Otherwise, line 32210 finds the most significant digit by dividing V (the decimal address) by 256 and storing it in UB (for upper byte), and the least significant one by subtracting 256 times UB from V. The result is PRINTed and subroutine 32400 is called for a pause.

6-BYTES TO DECIMAL ADDRESS

Line 32220 asks for the upper and lower bytes. Line 32230 rejects them if either one is outside the byte range. Otherwise, line 32240 multiplies the upper byte by 256 and adds it to the lower one. The result is PRINTed, and subroutine 32400 is called for a pause.

If you have a 4-digit hexadecimal number and want to find its decimal equivalence, think of it as two 2-digit hexadecimal numbers, the left being the most significant one. Using the hexadecimal to decimal conversion and then the byte to decimal, the equivalence can be found easily.

```
1  REM  ** CONVERSION **
10 TRUE = 1:FALSE = 0: DIM HX$(15): FOR I = 0 TO
     15: READ HX$(I): NEXT : DATA 0,1,2,3,4,5,6,7
     ,8,9,A,B,C,D,E,"F"
32000  HOME : VTAB 6: PRINT "1-DECIMAL TO BINARY"
     : PRINT : PRINT "2-BINARY TO DECIMAL": PRINT
     : PRINT "3-DECIMAL TO HEXADECIMAL": PRINT : PRINT
     "4-HEXADECIMAL TO DECIMAL"
32010  PRINT : PRINT "5-DECIMAL ADDRESS TO BYTES"
     : PRINT : PRINT "6-BYTES TO DECIMAL ADDRESS"
     : PRINT : PRINT "7-QUIT ";
32020  GET W$: IF W$ < "1" OR W$ > "7" THEN 32020

32030  ON  VAL (W$) GOTO 32040,32070,32100,32120,
     32200,32220,32280
```

```
32040  GOSUB 32380: IF FL THEN FL = FALSE: GOTO 3
     2040
32050 A$ = "": FOR I = 7 TO 0 STEP  - 1:S =  INT
     (V / 2 ^ I):V = V - S * 2 ^ I:A$ = A$ +  STR$
     (S): NEXT
32060  PRINT : PRINT : PRINT "  BINARY : "A$: GOSUB
     32400: GOTO 32000
32070  HOME : VTAB 10: INPUT "  BINARY:";W$: IF  LEN
     (W$) <  > 8 THEN  PRINT : PRINT "  8 DIGITS"
     : GOSUB 32400: GOTO 32070
32080 V = 0: FOR I = 0 TO 7:Z =  VAL ( MID$ (W$,8
      - I,1)): IF Z <  > 0 AND Z <  > 1 THEN  PRINT
     : PRINT "  WRONG DIGIT": GOSUB 32400: GOTO 3
     2070
32090 V = V + Z * 2 ^ I: NEXT : PRINT : PRINT "
     DECIMAL : "V: GOSUB 32400: GOTO 32000
32100  GOSUB 32380: IF FL THEN FL = FALSE: GOTO 3
     2100
32110 S =  INT (V / 16):A$ = HX$(S) + HX$(V - S *
     16): PRINT : PRINT "  HEXADECIMAL : "A$: GOSUB
     32400: GOTO 32000
32120  HOME : VTAB 10: INPUT "  HEXADECIMAL NUMBE
     R:";W$: IF  LEN (W$) <  > 2 THEN  PRINT : PRINT
     "  TWO DIGITS, PLEASE ": GOSUB 32400: GOTO 3
     2120
32130 V = 0: FOR I = 1 TO 2:X$ =  MID$ (W$,I,1): FOR
     J = 0 TO 15: IF X$ = HX$(J) THEN 32150
32140  NEXT : PRINT : PRINT "  WRONG DIGIT": GOSUB
     32400: GOTO 32000
32150 V = V + J * 16 ^ (2 - I): NEXT I: PRINT : PRINT
     "  DECIMAL : ";V: GOSUB 32400: GOTO 32000
32200  HOME : VTAB 10: INPUT "  DECIMAL NUMBER:";
     W$:V =  VAL (W$): IF V < 0 OR V > 65536 OR  INT
     (V) <  > V THEN  PRINT : PRINT "  0 <= X <=
     65536 , X INTEGER": GOSUB 32400: GOTO 32200
32210 UB =  INT (V / 256):LB = V - 256 * UB: PRINT
     : PRINT "  LOWER BYTE : "LB: PRINT : PRINT "
       UPPER BYTE : "UB: GOSUB 32400: GOTO 32000
32220  HOME : VTAB 8: INPUT "  LOWER BYTE : ";LB$
     : PRINT : INPUT "  UPPER BYTE : ";UB$:UB =  VAL
     (UB$):LB =  VAL (LB$)
32230  IF UB < 0 OR UB > 255 OR LB < 0 OR LB > 25
     5 OR  INT (LB) <  > LB OR  INT (UB) <  > UB THEN
      PRINT : PRINT "  WRONG VALUES": GOSUB 32400
     : GOTO 32220
32240 S = UB * 256 + LB: PRINT : PRINT "  DECIMAL
      : "S: GOSUB 32400: GOTO 32000
32280  HOME : END
32380  HOME : VTAB 10: INPUT "  DECIMAL NUMBER:";
     W$:V =  VAL (W$): IF V < 0 OR V > 255 OR  INT
     (V) <  > V THEN  PRINT : PRINT "  0 <= X <=2
     55 , X INTEGER": GOSUB 32400:FL = TRUE
```

```
32390  RETURN
32400  HTAB 2: VTAB 22: PRINT "  TYPE <RETURN> TO
      CONTINUE ";: GET W$: IF W$ <  >  CHR$ (13) THEN
     32400
32410  RETURN
```

11.14 SUBROUTINES THAT CALL THEMSELVES

When a GOSUB is found in a program, Applesoft stores the current line number (and where in the line number) and branches to the line in the GOSUB. Execution continues normally until a RETURN is found. Applesoft looks for the line number that it has stored and jumps back to it.

If there is a second GOSUB before the RETURN for the first one is found, Applesoft stores this line number too, and repeats the process for every new GOSUB; up to 25 subroutine addresses can be stored at any given time, with some limitations. (Unfinished FOR loops, as well as partial evaluation of expressions, take some of these locations.)

Thus, if a subroutine calls itself, Applesoft does exactly the same as it would for any line called.

At first sight, it might seem like the process would be stuck there, calling itself ad infinitum (or until the computer finds that more than 25 subroutines are "alive"). But if in one of those subroutines some condition is satisfied that makes the subroutine RETURN, the whole chain can be reversed, if things are done properly.

This process, called RECURSION, can be a very powerful tool, because certain operations are naturally recursive.

One example frequently used is the factorial operator. The usual way to define it is (we will only define it for the set of positive integers, excluding the zero):

$$N!=N\ast(N-1)\ast(N-2)\ast\ldots.\ast2\ast1$$

Another way to define it, using recursion, is:

$$\begin{array}{ll} N! = N \ast (N-1)! & \text{If } N>1, \\ \quad 1 & \text{If } N=1 \end{array}$$

Notice that without the special case in which N is assigned a definite value, the process would never end.

Let's see an example:

We want to compute 4!. Following the recursive definition,

4! = 4 * 3!
3! = 3 * 2!
2! = 2 * 1!
1! = 1 ;

therefore,

2! = 2 * 1 = 2
3! = 3 * 2 = 6
4! = 4 * 6 = 24

working backwards, just like a subroutine that returns to where it was called. The difference between the process carried out above and a regular subroutine is the fact that when we called it, the value of N was saved, and a different one was sent (when 4! = 4 * 3! called the ! subroutine to evaluate 3!, the 4 was temporarily stored, and N was sent with value 3).

Program RECURSIVE solves this problem using an array of N's instead of a single one.

If every time that the subroutine is called, an index is incremented, and every time there is a RETURN, the index is decremented, every level of subroutine will have its own N, and will be able to remember the value that is exclusively its own.

Line 10 initializes N ( ) so that it can be used to store the values at each level of subroutine. Line 30 initializes C to 1; it is going to be used as the index of N ( ), and 1 indicates that at the beginning, the program is in the first level of subroutine.

N (C) is entered, and rejected if it is outside the range (trying to calculate a number greater than 23 requires more than 23 subroutines, which Applesoft cannot handle). If it is accepted, line 40 calls subroutine 1000.

The first thing done by line 1000 is to check if the number whose factorial is to be calcualted is 1, the special case. If this is the case, F (the variable where the result of the calculation will be stored) is set to 1, the value of 1!, and the subroutine RETURNs.

If the number is not 1, line 1010 increments C and leaves N (C) intact, N (C–1)–1 is assigned to N (C) (the previous N, minus one), and subroutine 1000 is called.

While this happens, the subroutine is stuck calling itself without crossing the boundary of line 1010. When line 1000 finds the 1 and RETURNs, it decrements C. Line 1020 multiplies the current value of F (which, after the 1 is found, is exactly 1) by N (C), and line 1030 RETURNs to where it was called from, that is, once more to line 1020, until the RETURN found is not to line 1020 but to the main program. Then, F is equal to the factorial of the number typed.

Consider another example: In program BINARY SEARCH (Section 3.7), a particular value had to be looked for, in an array of sorted numbers. Three pointers were used: one points to the beginning of the array, one to the end, and one to the middle point, which is where the numbers were actually compared.

If the number is not found there, either the pointer to the beginning or the one pointing to the end of the array is changed, a new middle point is calculated, and the operation is repeated exactly the same way, this time considering not the whole array, but a smaller one going from the beginning point to the end one. So, written in English, what actually happens is:

1. Beginning point = 1; ending point = last index of the array. FIND THE NUMBER:
2. Middle point = point halfway between beginning and last points.
3. If the number at middle point is the one looked for, FOUND.
4. If the size of the array is 1: NOT FOUND. END.
5. If the number at middle point is bigger than the one looked for, Ending point = middle point–1; FIND THE NUMBER.
6. The number at middle point is smaller than the one looked for: Beginning point = middle point–1; FIND THE NUMBER.

As you can see, it is a naturally recursive process. Since the only values that vary are the beginning and ending points, those would be the only ones stored in an array; and since the list is reduced to less than half its original size every time the subroutine is called, the 25 levels of subroutine would be able to do a binary search in an array much bigger than the Apple can handle.

```
1  REM  ** RECURSIVE **
10  DIM N(100):C = 1
20  HOME
30 C = 1: PRINT : INPUT "NUMBER : ";N(C): IF N(C)
     < 1 OR N(C) > 23 OR  INT (N(C)) <  > N(C) THEN
     30
40  GOSUB 1000: PRINT : PRINT F: GOTO 30
1000  IF N(C) = 1 THEN F = 1:C = C - 1: RETURN
1010 C = C + 1:N(C) = N(C - 1) - 1: GOSUB 1000
1020 F = F * N(C):C = C - 1
1030  RETURN
```

11.15 EXTENDED PRECISION OPERATIONS

Applesoft uses variables that store up to nine digits, and any additional ones are chopped off. Sometimes, though, this precision is not enough, and there is no way to calculate even a trivial addition with numbers of more than nine digits.

Program EXTENDED PRECISION stores the numbers as strings, therefore allowing up to 255 digit accuracy; the operations are performed the same way as using a pen and pencil.

Ignoring negative numbers, the basic operations are described as follows.

Addition

1735 and 529 have to be added. They are placed one on top of the other:

```
1 7 3 5
  5 2 9
```

Add 5 and 9 (using the regular Applesoft addition, since only two digit accuracy is needed), getting 14. Since it is larger than 10, we subtract 10 from it, getting 4, and store 1 in a place that we will call the carry; 4 is the first digit of the sum.

Add 3 and 2, plus the carry, getting 6; perform the same procedure, this time without the carry; the sum is now 64.

The same procedure is used for as many digits as there are.

Subtraction

27 has to be subtracted from 93. They are placed one on top of the other:

9 3
2 7

Since 7 is bigger than 3, we add 10 to it and store 1 in the carry; 13 − 7 = 6, the first number. Now the carry is subtracted from the 9, and then the 2 to get 6. Result: 66.

Multiplication

27 and 93 have to be multiplied. We place them one on top of the other:

2 7
9 3

We multiply 3 by 7, getting 21, which becomes a 1 and a 2 in the carry. 3 times 2 is 6, plus the carry is 8, so the first line reads 81; the same operation is repeated with the 9, taking care to shift the result one place to the left; the two resulting numbers are then added.

Division

The procedure is the same as used with a pen and pencil, except for the case in which the divider has more than nine digits, because then we cannot use the Applesoft division. If that is the case, successive subtractions are performed, until the result is negative; the number of subtractions minus one is the quotient.

Both the multiplication and division need to use addition and subtraction; if negative numbers are allowed, all the operations become interdependent, and since there is only one subroutine for all the operations, it will have to call itself several times.

Table 11.4 shows how an addition can become a subtraction and vice versa, depending on the sign of the numbers used.

Table 11.4. Cases in addition and subtraction.

$A + B =$

$C + D$,	If $A > 0$ and $B > 0$;
$C - D$,	If $A > 0$ and $B < 0$;
$-(C - D)$,	If $A < 0$ and $B > 0$;
$-(C + D)$,	If $A < 0$ and $B < 0$.

$A - B =$

$C - D$,	If $A > 0$ and $B > 0$;
$C + D$,	If $A > 0$ and $B < 0$;
$-(C + D)$,	If $A < 0$ and $B > 0$;
$-(C - D)$,	If $A < 0$ and $B < 0$.

C = ABS (A): D = ABS (B)

No fractional part is allowed in these operations, but it is not a handicap if you think that the numbers are in fixed point notation mode, as opposed to floating point.

If you decide that your numbers will have, for example, five digits in the fractional part, instead of entering 1 you type its equivalent in the new convention: 100000. If you want to see it clearer, you add a decimal point or even commas at the thousand, million, etc., positions, just as done in Section 2.6.

Line 5 DIMensions M$ ( ), which is going to be used to store the partial sums in the multiplication; TRUE and FALSE get their Applesoft equivalents.

Line 10 asks for the numbers and the operation to be performed. Line 20 calls subroutine 1000.

Line 1005 sets three flags to FALSE; FL is going to say if the result of the operation is negative and F1 and F2 if the first and second numbers are negative. Line 1010 sets F1 and F2 if needed, and takes the signs off the numbers.

Line 1030 uses F1 and F2 to calculate IN, the code of the operation to perform, based on the four cases shown in Table 11.4.

Addition

The basic addition routine starts in line 1500. In line 1005, if the two numbers have less than eight digits, the operation becomes the usual Applesoft addition and the subroutine is terminated.

Otherwise, line 1510 sets SM$ (where the final result will be returned) to " " and C (the carry) to 0. After lines 1510 or 1520, depending on the length of the numbers, the longest will be in S1$, and the shortest (when they have the same length it does not make any difference) in S2$. If the length of the two numbers is different, line 1540 adds 0's until they have the same number of digits, which makes it possible to perform the operation using only one loop.

Line 1550 starts the II loop, which is going to take every couple of digits in S1$ and S2$, convert them to numbers, and add them. If there is any carry, it will be added too. If the result is greater than 10, line 1570 sets the carry to 1 and adjusts the number. It is then converted to its string representation and added to SM$.

At the end of the loop, in line 1590, if the carry is 1 it is added as a string to SM$.

Line 1600 RETURNs to the line that called it, and is branched to line 1120, where the negative symbol is added if necessary.

Line 1130 RETURNs, leaving subroutine 1000 definitely.

Subtraction

The initial procedure is similar to the addition, with line 1050 taking care of the different cases produced by the sign of the numbers; subroutine 2000 is called.

If the two numbers can be subtracted using the regular Applesoft operation, line 2020 performs it. If the result is found to be negative, FL is toggled (if the result was negative originally, it is now set to positive, and vice versa) and the absolute value of the result is used in line 2030 to return the string of the result.

If the numbers could not be processed regularly, line 2070 copies them into S1$ and S2$; SM$ (the final result) is emptied. If the second number is longer than the first one, subroutine 2300 swaps them and adjusts the sign in FL (A–B = –(B–A)). That way, the first number is longer than the second, and after lines 2090 and 2100, the length of the two are equal, having added 0's if necessary.

Now, having the same length, if a string comparison is made, it will be equivalent to a numeric comparison (string comparisons only correspond to numerical comparisons if the strings have the same length: "1" is smaller than "050", but "001" is not); if the first one is smaller (this time in value, not in size) than the second one, they are swapped by subroutine 2300 and FL is adjusted once more.

Line 2150 sets C, the carry, to 0, and starts the loop. In line 2160, the values of the individual digits are subtracted taking into consideration the carry; if the result is found to be negative, line 2170 adds 10 to it, and adjusts the carry. Line 2180 stores the string of every resulting digit in SM$. Subroutine 2100 gets rid of all the 0's at the left of the resulting number, and the subtraction subroutine terminates.

Line 1120 adds the sign if necessary, and RETURNs from subroutine 1000.

Multiplication

Lines 1005 through 1030 perform the same initialization done for the addition and subtraction. Line 1100 has an interesting feature: In multiplication and division, the result is negative only if one of the signs of the numbers is negative, but not both. This condition is called exclusive OR when working with boolean variables. Therefore, FL becomes TRUE (negative number) when (F1 OR F2), with the additional condition that (F1 AND F2) is FALSE, therefore NOT (F1 AND F2) is TRUE. Putting the two conditions together, the result is negative if (F1 OR F2) AND NOT (F1 AND F2).

Line 2502 performs the regular multiplication in Applesoft if the numbers have less than four digits each, to allow for a result with less than or exactly nine digits.

If they cannot be handled normally, line 2505 copies them into S3$ and S4$. Different variables than those used in the addition and subtraction subroutines are chosen, since those subroutines will be needed and called before the multiplication is completed.

The KK-loop started in line 2505 assigns to X$ one digit from

S3$. The carry is cleared, and line 2510 empties the corresponding line in M$ ( ), where the line produced by multiplying every digit of S4$ by X$ will be stored. If X$ is 0, there is no multiplication, and line 2510 jumps to 2580, to continue the KK-loop.

Since every line produced by multiplying S3$ by a digit of S4$ has to be shifted to the left, line 2520 adds as many 0's as are needed to the corresponding M$ ( ). Line 2530 starts getting the individual digits of S4$ and stores them in Y$. Line 2540 multiplies the two digits, and NC gets the carry, which unlike the addition and subtraction, here can be greater than 1. R is set to the residue, that is, the rightmost digit of the result.

Line 2550 adds the carry from the previous addition (in C) with R, and adjusts NC and T if the result is greater than 10.

Line 2560 adds the resulting digit to the previous contents of the corresponding line in M$ ( ), and sets C to NC. In line 2570, after a whole line has been processed, if there is a carry in C, it is added to M$ (KK).

Line 2590 copies the first line (M$ (1)) into TT$. If there is only one line, it is copied into SM$, and the multiplication subroutine is terminated. Otherwise, TT$ and the following M$ (KK)'s are added using subroutine 1500.

Line 1120 adjusts the sign, as usual.

Division

The sign flag is adjusted in the same way as for the multiplication; line 3005 copies this sign into F3, since the repetitive use of the subroutine 1000, called from here, will most likely destroy it. The numbers are copied into DI$ (dividend) and DV$ (divisor).

If the two numbers can be divided by Applesoft, line 3010 does it and the subroutine terminates. Otherwise, line 3030 starts the EE loop that gets individual digits from DI$. Line 3040 multiplies the contents of DD$ (the remainder, which at the beginning was set to 00, since the multiply part of subroutine 1000 does not work with null strings) by 10, using subroutine 10. This is equivalent to the digit-shift done to the remainder after each division cycle.

This number is then added to the digit just gotten from DI$, now in TP$ (again, subroutine 1000). If these smaller numbers to be divided can be handled by Applesoft, line 3050 divides them, line 3070 multiplies this partial quotient by the divider, and line 3080 subtracts it from TP$; the result is stored in DD$. Line 3100 adds V$ (the partial quotient) to the previous contents of WW$, the general quotient.

When the partial division cannot be handled by Applesoft, line 3060 jumps to subroutine 3200, which does repetitive subtractions until the result becomes negative; the quotient is then the number of subtractions minus 1. A limit of 1000 subtractions has been established, since more than this approaches eternity. A method to accelerate this subroutine is given below.

Line 3110 ends the EE-loop, and eliminates any nonsignificant 0's in the quotient. Line 3130 sets the sign flag to its value on entrance to subroutine 3000, and finally line 1120 takes care of the sign.

Since very large numbers can be the partial quotients of the intermediate divisions, subroutine 3200 might take forever to terminate. A pseudo-binary search could be devised, subtracting bigger and bigger numbers, maybe the divisor multiplied by 10 every time, until a closer limit is found. Then the process would be repeated starting with the original divisor, until the number is surrounded by two numbers one divisor apart.

```
1  REM  ** EXTENDED PRECISION **
2  DIM M$(255)
5 TRUE = 1:FALSE = 0: HOME
10  INPUT A$: INPUT B$: INPUT "1-ADD 2-SUBT 3-MUL
     T 4-DIV ";AS
20  GOSUB 1000: PRINT SM$: GOTO 10
1000  REM  ** CASE STUDY **
1005 FL = FALSE:F1 = FALSE:F2 = FALSE
1010  IF  LEFT$ (A$,1) = "-" THEN F1 = TRUE:A$ =
      RIGHT$ (A$, LEN (A$) - 1)
1020  IF  LEFT$ (B$,1) = "-" THEN F2 = TRUE:B$ =
      RIGHT$ (B$, LEN (B$) - 1)
1030 IN = 2 * F1 + F2: ON AS GOTO 1040,1050,1100,
     1110
1040  ON IN + 1 GOTO 1060,1070,1080,1090
1050  ON IN + 1 GOTO 1070,1060,1090,1080
1060 FL = FALSE: GOSUB 1500: GOTO 1120
1070 FL = FALSE: GOSUB 2000: GOTO 1120
1080 FL = TRUE: GOSUB 2000: GOTO 1120
1090 FL = TRUE: GOSUB 1500: GOTO 1120
1100 FL = (F1 OR F2) AND  NOT (F1 AND F2): GOSUB
     2500: GOTO 1120
1110 FL = (F1 OR F2) AND  NOT (F1 AND F2): GOSUB
     3000
```

```
1120  IF FL THEN SM$ = "-" + SM$
1130  RETURN
1500  REM  ** ADDITION ROUTINE **
1505  IF  LEN (A$) <  = 8 AND  LEN (B$) <  = 8 THEN
     SM$ =  STR$ ( VAL (A$) +  VAL (B$)): RETURN

1510 SM$ = "":S1$ = "":S2$ = "":C = 0: IF  LEN (B
     $) >  LEN (A$) THEN S1$ = B$:S2$ = A$: GOTO
     1530
1520 S1$ = A$:S2$ = B$
1530  IF  LEN (A$) =  LEN (B$) THEN 1550
1540  FOR II = 1 TO  LEN (S1$) -  LEN (S2$):S2$ =
     "0" + S2$: NEXT
1550  FOR II =  LEN (S2$) TO 1 STEP  - 1
1560 SM = C +  VAL ( MID$ (S1$,II,1)) +  VAL ( MID$
     (S2$,II,1))
1570 C = 0: IF SM > 9 THEN C = 1:SM = SM - 10
1580 SM$ =  STR$ (SM) + SM$
1590  NEXT : IF C = 1 THEN SM$ = "1" + SM$
1600  RETURN
2000  REM  ** SUBTRACTION ROUTINE **
2010  IF  LEN (A$) > 9 OR  LEN (B$) > 9 THEN 2070

2020 S =  VAL (A$) -  VAL (B$): IF S < 0 THEN FL =
      NOT FL
2030 SM$ =  STR$ ( ABS (S)): RETURN
2070 S1$ = A$:S2$ = B$:SM$ = "": IF  LEN (B$) >  LEN
     (A$) THEN  GOSUB 2300
2090  IF  LEN (S1$) =  LEN (S2$) THEN 2110
2100  FOR I = 1 TO  LEN (S1$) -  LEN (S2$):S2$ =
     "0" + S2$: NEXT
2110  IF S2$ > S1$ THEN  GOSUB 2300
2150  FOR I =  LEN (S1$) TO 1 STEP  - 1
2160 SM =  VAL ( MID$ (S1$,I,1)) -  VAL ( MID$ (S
     2$,I,1)) - C
2170 C = 0: IF SM < 0 THEN SM = SM + 10:C = 1
2180 SM$ =  STR$ (SM) + SM$: NEXT
2190  GOSUB 2210: RETURN
2210  FOR I = 1 TO  LEN (SM$): IF  MID$ (SM$,I,1)
      <  > "0" THEN SM$ =  RIGHT$ (SM$, LEN (SM$)
      - I + 1): RETURN
2220  NEXT : RETURN
2300 T$ = S1$:S1$ = S2$:S2$ = T$:FL =  NOT FL: RETURN

2500  REM  ** MULTIPLY ROUTINE **
2502  IF  LEN (A$) <  = 4 AND  LEN (B$) <  = 4 THEN
     SM$ =  STR$ ( VAL (A$) *  VAL (B$)): RETURN

2505 S3$ = A$:S4$ = B$: FOR KK =  LEN (S3$) TO 1 STEP
      - 1:X$ =  MID$ (S3$,KK,1):C = 0
2510 M$(KK) = "": IF X$ = "0" THEN 2580
2520  IF KK <  LEN (S3$) THEN  FOR LL = 1 TO  LEN
     (S3$) - KK:M$(KK) = M$(KK) + "0": NEXT
```

```
2530  FOR LL =  LEN (S4$) TO 1 STEP  - 1:Y$ =  MID$
     (S4$,LL,1)
2540 S =  VAL (X$) *  VAL (Y$):NC =  INT (S / 10)
     :R = S - NC * 10
2550 T = C + R: IF T > 9 THEN T = T - 10:NC = NC +
     1
2560 M$(KK) =  STR$ (T) + M$(KK):C = NC
2570  NEXT : IF C <  > 0 THEN M$(KK) =  STR$ (C) +
     M$(KK)
2580  NEXT
2590 TT$ = M$(1): IF  LEN (S3$) > 1 THEN  FOR KK =
     2 TO  LEN (S3$):A$ = TT$:B$ = M$(KK): GOSUB
     1500:TT$ = SM$: NEXT
2600 SM$ = TT$: RETURN
3000  REM  ** DIVISION ROUTINE **
3005 F3 = FL:WW$ = "":DI$ = A$:DV$ = B$:DD$ = "00
     "
3010  IF  LEN (A$) <  = 9 AND  LEN (B$) <  = 9 THEN
     SM$ =  STR$ ( INT ( VAL (A$) /  VAL (B$))): RETURN
3030  FOR EE = 1 TO  LEN (DI$):TP$ =  MID$ (DI$,E
     E,1)
3040 A$ = DD$:B$ = "10":AS = 3: GOSUB 1000:AS = 1
     :A$ = SM$:B$ = TP$: GOSUB 1000:TP$ = SM$
3050  IF  LEN (TP$) <  = 9 AND  LEN (DV$) <  = 9 THEN
     V$ =  STR$ ( INT ( VAL (TP$) /  VAL (DV$))):
      GOTO 3070
3060 A$ = TP$:B$ = DV$: GOSUB 3200:V$ = SM$
3070 A$ = V$:B$ = DV$:AS = 3: GOSUB 1000
3080 A$ = TP$:B$ = SM$:AS = 2: GOSUB 1000
3090 DD$ = SM$
3100 WW$ = WW$ + V$
3110  NEXT :SM$ = WW$: FOR II = 1 TO  LEN (SM$): IF
      MID$ (SM$,II,1) <  > "0" THEN SM$ =  RIGHT$
     (SM$, LEN (SM$) - II + 1): GOTO 3130
3120  NEXT :SM$ = "0"
3130 FL = F3: RETURN
3200 S5$ = A$:S6$ = B$: FOR UU = 1 TO 1000:A$ = S
     5$:B$ = S6$:AS = 2: GOSUB 1000: IF FL THEN S
     M$ =  STR$ (UU - 1): RETURN
3210 S5$ = SM$: NEXT : STOP
```

11.16 LOWER CASE FOR THE PRINTER

Since the Apple cannot display lower case characters, the shift key does not work and all the letters enter as upper case.

The ASCII code of the lower case letters corresponds to the ASCII code of the same letter in upper case, plus 32.

Therefore, if an upper case character is converted into its

ASCII equivalent using the function ASC, 32 is added to this number and then it is returned to its character form with CHR$, the character will be in lower case, provided it is a letter; other characters will become different symbols.

Program LOWER CASE is a very elementary text editor (the most elementary), and it was written only to show you how text can be produced in lower case in the Apple.

Line 10 sets D$ to be used in the disk operations, and DIMensions T$ ( ), which is where the text will be stored. Line 20 shows the following menu:

1-TYPE
2-VIEW
3-LOAD
4-SAVE
5-PROCESS
6-PRINT
7-QUIT

1-TYPE

Line 50 PRINTs a message informing you how to return to the menu. Lines 60 and 70 enter the text using a simple INPUT statement. If the word entered is END, N is adjusted to the last value of I, and the program jumps back to the menu.

2-VIEW

Line 180 PRINTs a message saying that those characters shown in INVERSE correspond to upper case (normal ones are lower case) and the I-loop examines every text line (T$ (I)). If T$ (I) doesn't have any characters, it examines the next line. Otherwise, a new loop examines every character in the current line. If the character is found to be an upper case letter (the ASCII code for upper case letters is SMALLER THAN 97), it is PRINTed in INVERSE; otherwise, it is converted to upper case (because in the Apple the lower case letters are not shown as letters at all in the screen), and PRINTed in NORMAL, which is the default mode. That way, you will be able to tell how your text will look when it is PRINTed in a printer that supports lower case.

5-PROCESS

Line 120 confirms the option. Lines 130 through 170 perform exactly the same action as the VIEW option, except that, instead of PRINTing it in NORMAL or INVERSE, it actually reforms the character (adds 32 to its ASCII code), so that at the end of every loop, T$ (I) will have all its letters in lower case (the other symbols are not touched).

6-PRINT

Sending T$ ( ) to the printer, you can see that the text has actually become lower case.

As discussed previously, this is the simplest version of a word processor, yet it shows you how to do something that is normally impossible in the Apple. If you find this option useful, you can add features like selection of what lines are to be downshifted, which characters, etc.

```
1  REM  ** LOWER CASE **
10 D$ =  CHR$ (4): DIM T$(100)
20  HOME : VTAB 6: PRINT "1-TYPE": PRINT : PRINT
     "2-VIEW": PRINT : PRINT "3-LOAD": PRINT : PRINT
     "4-SAVE": PRINT : PRINT "5-PROCESS": PRINT :
      PRINT "6-PRINT": PRINT : PRINT "7-QUIT ";
30  GET W$: IF W$ < "1" OR W$ > "7" THEN 30
40  ON  VAL (W$) GOTO 50,180,80,100,120,250,300
50  HOME : PRINT "TYPE ";: INVERSE : PRINT "END <
     RETURN>";: NORMAL : PRINT " TO END": PRINT
60  FOR I = 1 TO 100: INPUT "";W$: IF W$ = "END" THEN
     N = I - 1: GOTO 20
70 T$(I) = W$: NEXT :N = 100: GOTO 20
80  GOSUB 200: PRINT D$"READ "FL$: INPUT N
90  FOR I = 1 TO N: INPUT T$(I): NEXT : PRINT D$"
     CLOSE": GOTO 20
100  GOSUB 200: PRINT D$"WRITE "FL$: PRINT N
110  FOR I = 1 TO N: PRINT T$(I): NEXT : PRINT D$
     "CLOSE": GOTO 20
120  PRINT : PRINT : INVERSE : HTAB 15: PRINT "PR
     OCESSING": NORMAL
130  FOR I = 1 TO N:X$ = "": IF  LEN (T$(I)) = 0 THEN
     170
140  FOR J = 1 TO  LEN (T$(I)):W$ =  MID$ (T$(I),
     J,1): IF W$ >  = "A" AND W$ <  = "Z" THEN X$
      = X$ +  CHR$ ( ASC (W$) + 32): GOTO 160
150 X$ = X$ + W$
```

```
160   NEXT J:T$(I) = X$
170   NEXT I: GOTO 20
180   HOME : PRINT "INVERSE IS UPPER CASE": PRINT
      : FOR I = 1 TO N: IF  LEN (T$(I)) = 0 THEN 1
      95
185   FOR J = 1 TO  LEN (T$(I)):W$ =  MID$ (T$(I),
      J,1): IF W$ <  CHR$ (97) THEN  INVERSE : PRINT
      W$;: NORMAL : GOTO 192
190   PRINT  CHR$ ( ASC (W$) - 32);
192   NEXT J
195   PRINT : NEXT I: PRINT : PRINT "TYPE <RETURN>
       TO CONTINUE ";
197   GET W$: IF W$ <  > CHR$ (13) THEN 197
198   GOTO 20
200   PRINT : PRINT : INPUT "FILENAME : ";FL$: IF
      FL$ = "" THEN  POP : GOTO 20
210 W$ =  LEFT$ (FL$,1): IF W$ < "A" OR W$ > "Z" THEN
      200
220   PRINT D$"OPEN "FL$: RETURN
250   HOME : VTAB 9: PRINT "TYPE <RETURN> WHEN THE
      ": PRINT : PRINT "PRINTER IS READY ";
260   GET W$: IF W$ <  > CHR$ (13) THEN 260
270   PRINT : PRINT D$"PR#1": PRINT  CHR$ (9)"80N"

280   FOR I = 1 TO N: PRINT T$(I): NEXT : PRINT D$
      "PR#0": GOTO 20
300   HOME : END
```

INDEX

Activity switch 78
Alphabet 164, 169, 175
Animation 149
Amplifying sounds 183
Attribute 82

Backspace 2, 3
Binary file 217
Binary numbers 59, 220, 225
Binary search 40, 47
Boolean variables 201
Bubble sort 32, 33, 38
Burp 181, 182

Cassette output 183
Catalog 62, 87
Character 2, 74, 209, 213
Click 181
Color 115, 117, 119, 120
Command, hidden 206
Create a program 189
Character, unwanted 2
Control-C 1

Data creator 66
Data packing 58
Default 8
Delete 9, 10, 53
DOS 62
Draw 90, 117, 155, 157
Drive 60
Duration 183, 192, 197

Editor, text 74, 96
Editor, low resolution 117
Editor, high resolution 125
Emulator 172
Equivalences 223
Escape 12
Exec files 63
Extended precision 230
Extra ignored 51

Files 51, 53, 55
Files, exec 63
Files, sequential 51, 54
Files, random access 54

Fixing length 55
Format 21, 22, 23, 26, 28

Garbage collection 219
Graphics 7
GR 77, 115
GET 2, 15, 19, 52, 53

Harmony 196
HCOLOR 77, 79, 126, 138, 147, 149, 158
Help 93
Hexadecimal 222
Hidden 206, 208, 209, 213
High resolution 77, 78, 80, 125
HIMEM 61, 62
HGR 65, 77

Input 1, 7, 15, 17, 51, 54, 55
Insertion sort 34
Inverse 171

Keyboard 15, 17, 18, 19, 61

Length 5, 7, 23, 29, 55
Line, prompt 6
Linked list 42, 43
Low resolution 77, 80, 88, 115
Lower case 238

Maxfiles 61
Menu 12
Mode 89
Music entry 191
Negative 143
NEW 204
Noise 182
Notes, musical 183, 184, 186
Numbers 4

Output 21
Overflow 2

Pack data 58
Pages 75, 93, 95, 124, 125, 129, 131, 146
Pitch 181, 192, 197
Play 74, 188
Plotting 63
Pointers 42
Precision, extended 230
Printer, lower case 238
Programs, relocating 215
Prompt line 6

Random access files 54
Rattle 182
Record 55
Recuperate line 10
Recursion 227
Relocate program 215
Reset 218
Resolution, high 77, 78, 80, 125
Resolution, low 77, 80, 88, 115
Return 4, 54
ROT 155, 164
Rounding 25

Scale (HGR) 155
Scale, musical 184
Scanner 132, 137

Screen 75, 115
Scrolling 85, 116
Search, binary 40, 47
Selective input 13
Separators 52
Sequential files 51, 54
Shapes 150, 151, 156
Shell sort 35
Siren 183
Slot 60
Sort, Bubble 32, 33, 38
Sort, insertion 34
Sorting 31
Sound 181
Switch 75, 78, 79

Text 77, 79, 80, 81, 93, 95, 115, 133, 163, 169

Utilities 201

Variables, boolean 201
Volume 60

$14.95

FANCY PROGRAMMING IN APPLESOFT®

Gabriel Cuellar

Discover the hidden secrets of Applesoft BASIC! **Fancy Programming in Applesoft** gives you the tips, the techniques, and the shortcuts to give your programs that professional touch. You'll find both programs and complete subroutines like: BACKSPACE, LENGTH CONTROL, BUBBLE SORT, SCREEN DUMP, HIGH RESOLUTION EDITOR, SHAPE GENERATOR. Along with these you'll find out—

- How to recover a program if you accidentally type NEW
- How to INPUT data without interrupting or stopping the program as it runs
- How to use more than two pages of low resolution
- How to save an Applesoft program as a binary file.

Fancy Programming in Applesoft is a treasure chest of information for getting the most out of Applesoft BASIC and DOS. So before you jump into the monitor and Assembly Language to do things you thought were impossible in Applesoft BASIC, take a look through Gabriel Cuellar's guide to the world of fancy programming.

*APPLESOFT is a trademark of Apple Computer Inc.

RESTON PUBLISHING COMPANY, INC.
A Prentice-Hall Company
Reston, Virginia

0-8359-1856-4

cover design: Joyce Thompson cover illustration: Donna Ward

Cuellar

Fancy Programming in Applesoft

Reston